SCA Policy Framework Version 1.1

Committee Draft 03/Public Review 02

5 May 2010

Specification URIs:
This Version:
http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd03.html
http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd03.doc
http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd03.pdf (Authoritative)
Previous Version:
http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-02.html
http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-02.doc
http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1-spec-cd-02.pdf (Authoritative)
Latest Version:
http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1.html
http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1.doc
http://docs.oasis-open.org/opencsa/sca-policy/sca-policy-1.1.pdf (Authoritative)

Technical Committee:
OASIS SCA Policy TC
Chair(s):
David Booz, IBM <booz@us.ibm.com>
Ashok Malhotra, Oracle <ashok.malhotra@oracle.com>
Editor(s):

David Booz, IBM <booz@us.ibm.com>
Michael J. Edwards, IBM <mike_edwards@uk.ibm.com>
Ashok Malhotra, Oracle <ashok.malhotra@oracle.com>

Related work:
This specification replaces or supercedes:

- SCA Policy Framework Specification Version 1.00 March 07, 2007

This specification is related to:
OASIS Committee Draft 05, “SCA Assembly Model Specification Version 1.1”, January 2010.
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd05.pdf

Declared XML Namespace(s):
In this document, the namespace designated by the prefix “sca” is associated with the namespace URL docs.oasis-open.org/ns/opencsa/sca/200912**.** This is also the default namespace for this document.

Abstract:

TBD

Status:

Notices

Table of Contents

1

1 Introduction

The capture and expression of non-functional requirements is an important aspect of service definition and has an impact on SCA throughout the lifecycle of components and compositions. SCA provides a framework to support specification of constraints, capabilities and QoS expectations from component design through to concrete deployment. This specification describes the framework and its usage.

Specifically, this section describes the SCA policy association framework that allows policies and policy subjects specified using WS-Policy [WS-Policy] and WS-PolicyAttachment [WS-PolicyAttach], as well as with other policy languages, to be associated with SCA components.

This document should be read in conjunction with the SCA Assembly Specification [SCA-Assembly]. Details of policies for specific policy domains can be found in sections 7, 8 and 9.

1.1 Terminology

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

1.2 XML Namespaces

Prefixes and Namespaces used in this Specification

Prefix	XML Namespace	Specification
`sca`	`docs.oasis-open.org/ns/opencsa/sca/200912~~200903~~` This is assumed to be the default namespace in this specification. xs:QNames that appear without a prefix are from the SCA namespace.	[SCA-Assembly]
`acme`	Some namespace; a generic prefix	
`wsp`	`http://www.w3.org/2006/07/ws-policy`	[WS-Policy]
`xs`	`http://www.w3.org/2001/XMLSchema`	[XML Schema Datatypes]

Table 1-1: XML Namespaces and Prefixes

1.3 Normative References

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

[SCA-Assembly] OASIS Committee Draft 05~~03~~, “Service Component Architecture Assembly Model Specification Version 1.1”, January 2010~~March 2009~~.
http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-1.1-spec-cd05~~cd03~~.pdf

[SCA-Java-Annotations]
OASIS Committee Draft 04~~02~~, “SCA Java Common Annotations and APIs Specification Version 1.1”, February 2010~~2009~~.
http://docs~~www~~.oasis-open.org/opencsa/sca-j~~committees/download.php/31427~~/sca-javacaa-1.1-spec-cd04~~cd02~~.pdf

[SCA-WebServicesBinding]
OASIS Committee Draft 03~~01~~, “SCA Web Services Binding Specification Version 1.1”, July 2009~~August 2008~~.
http://docs.oasis-open.org/opencsa/sca-bindings/sca-wsbinding-1.1-spec-cd03~~cd01~~.pdf

[WSDL] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language – Appendix http://www.w3.org/TR/2006/CR-wsdl20-20060327/

[WS-AtomicTransaction]
Web Services Atomic Transaction (WS-AtomicTransaction)
http://docs.oasis-open.org/ws-tx/wsat/2006/06.

[WSDL-Ids] SCA WSDL 1.1 Element Identifiers – forthcoming W3C Note
http://dev.w3.org/cvsweb/~checkout~/2006/ws/policy/wsdl11elementidentifiers.html

[WS-Policy] Web Services Policy (WS-Policy)
http://www.w3.org/TR/ws-policy

[WS-PolicyAttach] Web Services Policy Attachment (WS-PolicyAttachment)
http://www.w3.org/TR/ws-policy-attachment

[XPATH] XML Path Language (XPath) Version 1.0.
http://www.w3.org/TR/xpath

[XML-Schema2] XML Schema Part 2: Datatypes Second Edition XML Schema Part 2: Datatypes Second Edition, Oct. 28 2004.
http://www.w3.org/TR/xmlschema-2/

1.4 Naming Conventions

This specification follows some naming conventions for artifacts defined by the specification, as follows:

~~1.~~ For the names of elements and the names of attributes within XSD files, the names follow the CamelCase convention, with all names starting with a lower case letter, e.g. <element name="policySet" type="…"/>.

~~2.~~ For the names of types within XSD files, the names follow the CamelCase convention with all names starting with an upper case letter, e.g. <complexType name="PolicySet">.

~~3.~~ For the names of intents, the names follow the CamelCase convention, with all names starting with a lower case letter, EXCEPT for cases where the intent represents an established acronym, in which case the entire name is in upper case. An example of an intent which is an acronym is the "SOAP" intent.

2 Overview

2.1 Policies and PolicySets

The term ***Policy*** is used to describe some capability or constraint that can be applied to service components or to the interactions between service components represented by services and references. An example of a policy is that messages exchanged between a service client and a service provider have to be encrypted, so that the exchange is confidential and cannot be read by someone who intercepts the messages.

In SCA, services and references can have policies applied to them that affect the form of the interaction that takes place at runtime. These are called ***interaction policies***.

Service components can also have other policies applied to them, which affect how the components themselves behave within their runtime container. These are called ***implementation policies***.

How particular policies are provided varies depending on the type of runtime container for implementation policies and on the binding type for interaction policies. Some policies can be provided as an inherent part of the container or of the binding – for example a binding using the https protocol will always provide encryption of the messages flowing between a reference and a service. Other policies can optionally be provided by a container or by a binding. It is also possible that some kinds of container or kinds of binding are incapable of providing a particular policy at all.

In SCA, policies are held in ***policySets***, which can contain one or many policies, expressed in some concrete form, such as WS-Policy assertions. Each policySet targets a specific binding type or a specific implementation type. PolicySets are used to apply particular policies to a component or to the binding of a service or reference, through configuration information attached to a component or attached to a composite.

For example, a service can have a policy applied that requires all interactions (messages) with the service to be encrypted. A reference which is wired to that service needs to support sending and receiving messages using the specified encryption technology if it is going to use the service successfully.

In summary, a service presents a set of interaction policies, which it requires the references to use. In turn, each reference has a set of policies, which define how it is capable of interacting with any service to which it is wired. An implementation or component can describe its requirements through a set of attached implementation policies.

2.2 Intents describe the requirements of Components, Services and References

SCA ***intents*** are used to describe the abstract policy requirements of a component or the requirements of interactions between components represented by services and references. Intents provide a means for the developer and the assembler to state these requirements in a high-level abstract form, independent of the detailed configuration of the runtime and bindings, which involve the role of application deployer. Intents support late binding of services and references to particular SCA bindings, since they assist the deployer in choosing appropriate bindings and concrete policies which satisfy the abstract requirements expressed by the intents.

It is possible in SCA to attach policies to a service, to a reference or to a component at any time during the creation of an assembly, through the configuration of bindings and the attachment of policy sets. Attachment can be done by the developer of a component at the time when the component is written or it can be done later by the deployer at deployment time. SCA recommends a late binding model where the bindings and the concrete policies for a particular assembly are decided at deployment time.

SCA favors the late binding approach since it promotes re-use of components. It allows the use of components in new application contexts, which might require the use of different bindings and different concrete policies. Forcing early decisions on which bindings and policies to use is likely to limit re-use and limit the ability to use a component in a new context.

For example, in the case of authentication, a service which requires the client to be authenticated can be marked with an intent called "**clientAuthentication**". This intent marks the service as requiring the client to be authenticated without being prescriptive about how it is achieved. At deployment time, when a ~~the~~ binding is chosen for the service (say SOAP over HTTP), the deployer can apply suitable policies to the service which provide aspects of WS-Security and which supply a group of one or more authentication technologies.

In many ways, intents can be seen as restricting choices at deployment time. If a service is marked with the **confidentiality** intent, then the deployer has to use a binding and a policySet that provides for the encryption of the messages.

The set of intents available to developers and assemblers can be extended by policy administrators. The SCA Policy Framework specification does define a set of intents which address the infrastructure capabilities relating to security, transactions and reliable messaging.

2.3 Determining which policies apply to a particular wire

Multiple policies can be attached to both services and to references. Where there are multiple policies, they can be organized into policy domains, where each domain deals with some particular aspect of the interaction. An example of a policy domain is confidentiality, which covers the encryption of messages sent between a reference and a service. Each policy domain can have one or more policy. Where multiple policies are present for a particular domain, they represent alternative ways of meeting the requirements for that domain. For example, in the case of message integrity, there could be a set of policies, where each one deals with a particular security token to be used: e.g. X509, SAML, Kerberos. Any one of the tokens can be used - they will all ensure that the overall goal of message integrity is achieved.

In order for a service to be accessed by a wide range of clients, it is good practice for the service to support multiple alternative policies within a particular domain. So, if a service requires message confidentiality, instead of insisting on one specific encryption technology, the service can have a policySet which has a number of alternative encryption technologies, any of which are acceptable to the service. Equally, a reference can have a policySet attached which defines the range of encryption technologies which it is capable of using. Typically, the set of policies used for a given domain will reflect the capabilities of the binding and of the runtime being used for the service and for the reference.

When a service and a reference are wired together, the policies declared by the policySets at each end of the wire are matched to each other. SCA does not define how policy matching is done, but instead delegates this to the policy language (e.g. WS-Policy) used for the binding. For example, where WS-Policy is used as the policy language, the matching procedure looks at each domain in turn within the policy sets and looks for 1 or more policies which are in common between the service and the reference. When only one match is found, the matching policy is used. Where multiple matches are found, then the SCA runtime can choose to use any one of the matching policies. No match implies that the configuration is not valid and the deployer needs to take an action.

3 Framework Model

The SCA Policy Framework model is comprised of ***intents*** and ***policySets***. Intents represent abstract assertions and Policy Sets contain concrete policies that can be applied to SCA bindings and implementations. The framework describes how intents are related to policySets. It also describes how intents and policySets are utilized to express the constraints that govern the behavior of SCA bindings and implementations. Both intents and policySets can be used to specify QoS requirements on services and references.

The following section describes the Framework Model and illustrates it using Interaction Policies. Implementation Policies follow the same basic model and are discussed later in section 1.5.

3.1 Intents

As discussed earlier, an ***intent*** is an abstract assertion about a specific Quality of Service (QoS) characteristic that is expressed independently of any particular implementation technology. An intent is thus used to describe the desired runtime characteristics of an SCA construct. Typically, intents are defined by a policy administrator. See section [Policy Administrator] for a more detailed description of SCA roles with respect to Policy concepts, their definition and their use. The semantics of an intent can not always be available normatively, but could be expressed with documentation that is available and accessible.

For example, an intent named **integrity** can be specified to signify that communications need to be protected from possible tampering. This specific intent can be declared as a requirement by some SCA artifacts, e.g. a reference. Note that this intent can be satisfied by a variety of bindings and with many different ways of configuring those bindings. Thus, the reference where the intent is expressed as a requirement could eventually be wired using either a web service binding (SOAP over HTTP) or with an EJB binding that communicates with an EJB via RMI/IIOP.

Intents can be used to express requirements for ***interaction policies*** or ***implementation policies***. The **integrity** intent in the above example is used to express a requirement for an interaction policy. Interaction policies are, typically, applied to a *service* or *reference*. They are meant to govern the communication between a client and a service provider. Intents can also be applied to SCA component implementations as requirements for ***implementation policies***. These intents specify the qualities of service that need to be provided by a container as it runs the component. An example of such an intent could be a requirement that the component needs to run in a transaction.

If the configured instance of a binding is in conflict with the intents and policy sets selected for that instance, the SCA runtime MUST raise an error. [POL30001]. For example, a web service binding which requires the SOAP intent but which points to a WSDL binding that does not specify SOAP.

For convenience and conciseness, it is often desirable to declare a single, higher-level intent to denote a requirement that could be satisfied by one of a number of lower-level intents. For example, the **confidentiality** intent requires either message-level encryption or transport-level encryption.

Both of these are abstract intents because the representation of the configuration necessary to realize these two kinds of encryption could vary from binding to binding, and each would also require additional parameters for configuration.

An intent that can be completely satisfied by one of a choice of lower-level intents is referred to as a *qualifiable intent*. In order to express such intents, the intent name can contain a qualifier: a "." followed by a *xs:string* name. An intent name that includes a qualifier in its name is referred to as a *qualified intent*, because it is "qualifying" how the qualifiable intent is satisfied. A qualified intent can only qualify one qualifiable intent, so the name of the qualified intent includes the name of the qualifiable intent as a prefix, for example, **clientAuthentication.message**.

In general, SCA allows the developer or assembler to attach multiple qualifiers for a single qualifiable intent to the same SCA construct. However, domain-specific constraints can prevent the use of some combinations of qualifiers (from the same qualifiable intent).

Intents, their qualifiers and their defaults are defined using the ~~following~~ pseudo schema in Snippet 3-1:

```
<intent  name="xs:NCName"
         constrains ="list of QNames"?
         requires="list of QNames"?
         excludes="list of QNames"?
         mutuallyExclusive="boolean"?
         intentType="xs:string"? >
   <description> xs:string.</description>?
   <qualifier name = "xs:string"  default = "xs:boolean" ?>*
         <description> xs:string.</description>?
   </qualifier>
</intent>
```

Snippet 3-1: intent Pseudo-Schema

Where the intent element has the following attributes:

- @name (1..1) - an NCName that defines the name of the intent. The QName for an intent MUST be unique amongst the set of intents in the SCA Domain. [POL30002]

- @constrains (0..1) - a list of QNames that specifies the SCA constructs that this intent is meant to configure. If a value is not specified for this attribute then the intent can apply to any SCA element.

  Note that the "constrains" attribute can name an abstract element type, such as sca:binding in our running example. This means that it will match against any binding used within an SCA composite file. An SCA element can match @constrains if its type is in a substitution group.

- @requires (0..1) - contains a list of QNames~~Qnames~~ of intents which defines the set of all intents that the referring intent requires. In essence, the referring intent requires all the intents named to be satisfied. This attribute is used to compose an intent from a set of other intents. Each QName in the @requires attribute MUST be the QName of an intent in the SCA Domain. [POL30015] This use is further described in Profile Intents~~Section 3.3. below.~~

@excludes (0..1) - a list of QNames of intents that cannot be used with this intent. Intents might describe a policy that is incompatible or otherwise unrealizable when specified with other intents, and therefore are considered to be mutually exclusive. Each QName in the @excludes attribute MUST be the QName of an intent in the SCA Domain. [POL30016]

Two intents are mutually exclusive when any of the following are true:

One of the two intents lists the other intent in its @excludes list.

- Both intents list the other intent in their respective @excludes list.
  If only one qualifier for an intent is given it MUST be used as the default qualifier for the intent.

~~[POL30023]~~

Where one intent is attached to an element of an SCA composite and another intent is attached to one of the element's parents, the intent(s) that are effectively attached to the element differs depending on whether the two intents are mutually exclusive (see @excludes above and "~~section~~ Attaching intents to SCA elements".

Usage of @requires attribute for specifying intents).

- @mutuallyExclusive (0..1) - a boolean with a default of "false". If this attribute is present and has a value of "true" it indicates that the qualified intents defined for this intent are mutually exclusive.

- @intentType attribute (0..1) defines whether the intent is an interaction intent or an implementation intent. A value of "interaction", which is the default value, indicates that the intent is an interaction intent. A value of "implementation" indicates that the intent is an implementation intent.

One or more <qualifier> child elements can be used to define qualifiers for the intent. The attributes of the qualifier element are:

- @name (1..1) - declares the name of the qualifier. The name of each qualifier MUST be unique within the intent definition. [POL30005].
- @default (0..1) - a boolean value with a default value of "false". If @default="true" the particular qualifier is the default qualifier for the intent. If an intent has more than one qualifier, one and only one MUST be declared as the default qualifier. [POL30004]. If only one qualifier for an intent is given it MUST be used as the default qualifier for the intent. [POL30025]~~.~~
- qualifier/description (0..1) - an xs:string that holds a textual description of the qualifier.

For example, the **confidentiality** intent which has qualified intents called **confidentiality.transport** and **confidentiality.message** can~~may~~ be defined as:

```
<intent name="confidentiality" constrains="sca:binding">
   <description>
      Communication through this binding must prevent
      unauthorized users from reading the messages.
```

```
   ~~</description>~~
   ~~<qualifier name="transport">~~
      ~~<description>Automatic encryption by transport~~
```

```
   </description>
   <~~</~~qualifier~~ name="transport"~~>
      <description>Automatic encryption by transport
      </description>
   </qualifier>
   <qualifier name="message" default='true'>
      <description>Encryption applied to each message
      </description>
   </qualifier>
</intent>
```

Snippet 3-2: Example intent Definition

An Intent can be contributed to~~All~~ the ~~intents in a~~ SCA Domain by including its definition in a ~~are defined in a global, domain-wide file named~~ definitions.xml file within a Contribution in the Domain.~~.~~ Details of the definitions.xml files~~this file~~ are described in the SCA Assembly Model [SCA-Assembly].

SCA normatively defines a set of core intents that all SCA implementations are expected to support, to ensure a minimum level of portability. Users of SCA can define new intents, or extend the qualifier set of existing intents. SCA implementations supporting both Direct Attachment and External Attachment mechanisms MUST ignore policy sets applicable to any given SCA element via the Direct Attachment mechanism when there exist policy sets applicable to the same SCA element via the External Attachment mechanism [POL30024] It is also good practice for the Domain to include concrete policies which satisfy these intents (this may be achieved through the provision of appropriate binding types and implementation types, augmented by policy sets that apply to those binding types and implementation types).

The normatively defined intents in the SCA specification might evolve in future versions of this specification. New intents could be added, additional qualifiers could be added to existing intents and the default qualifier for existing intents could change. Such changes would cause the namespace for the SCA specification to change.

3.2 Interaction Intents and Implementation Intents

An interaction intent is an intent designed to influence policy which applies to a service, a reference and the wires that connect them. Interaction intents affect wire matching between the two ends of a wire and/or the set of bytes that flow between the reference and the service when a service invocation takes place.

Interaction intents typically apply to <binding/> elements.

An implementation intent is an intent designed to influence policy which applies to an implementation artifact or to the relationship of that artifact to the runtime code which is used to execute the artifact. Implementation intents do not affect wire matching between references and services, nor do they affect the bytes that flow between a reference and a service.

Implementation intents often apply to <implementation/> elements, but they can also apply to <binding/> elements, where the desire is to influence the activity of the binding implementation code and how it interacts with the remainder of the runtime code for the implementation.

Interaction intents and implementation intents are distinguished by the value of the @intentType attribute in the intent definition.

3.3 Profile Intents

An intent that is satisfied only by satisfying *all* of a set of other intents is called a **profile intent**. It can be used in the same way as any other intent.

The presence of @requires attribute in the intent definition signifies that this is a profile intent. The @requires attribute can include all kinds of intents, including qualified intents and other profile intents.

However, while a profile intent can include qualified intents, it cannot be a qualified intent. Thus, the name of a profile intent MUST NOT have a "." in it. [POL30006]

Requiring a profile intent is semantically identical to requiring the list of intents that are listed in its @requires attribute. If a profile intent is attached to an artifact, all the intents listed in its @requires attribute MUST be satisfied as described in section 0. [POL30007]

An example of a profile intent is an intent called **messageProtection** which is a shortcut for specifying both **confidentiality** and **integrity**, where **integrity** means to protect against modification, usually by signing. The intent definition is shown in Snippet 3-3~~looks like the following~~:

```
<intent name="messageProtection"
   constrains="sca:binding"
   requires="confidentiality integrity">
   <description>
      Protect messages from unauthorized reading or modification.
   </description>
</intent>
```

Snippet 3-3: Example Profile Intent

3.4 PolicySets

A ***policySet*** element is used to define a set of concrete policies that apply to some binding type or implementation type, and which correspond to a set of intents provided by the policySet.

The pseudo schema for policySet is shown in Snippet 3-4~~below~~:

```
<policySet name="NCName"
         provides="listOfQNames"?
         appliesTo="xs:string"?
         attachTo="xs:string"?
         xmlns=http://docs.oasis-
open~~www.osoa~~.org/ns/opencsa~~xmlns~~/sca/200912~~1.0~~
         xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
   <policySetReference name="xs:QName"/>*
   <intentMap/>*
   <xs:any>*
</policySet>
```

Snippet 3-4: policySet Pseudo-Schema

PolicySet has the ~~following~~ attributes:

- @name (1..1) - the name for the policySet. The value of the @name attribute is the local part of a QName. The QName for a policySet MUST be unique amongst the set of policySets in the SCA Domain. [POL30017]

@appliesTo (0..1) - a string which is an XPath 1.0 expression identifying one or more SCA constructs this policySet can configure. The contents of @appliesTo MUST match the XPath 1.0 [XPATH] production *Expr.* [POL30018] The @appliesTo attribute uses the "Deployed Composites Infoset~~ for External Attachment~~" as described in Appendix A: The Deployed Composites Infoset section.

- Section 0 "".

- @attachTo (0..1) - a string which is an XPath 1.0 expression identifying one or more elements in the Domain. It is used to declare which set of elements the policySet is actually attached to. The contents of @attachTo MUST match the XPath 1.0 production Expr. [POL30019] The XPath value of the @attachTo attribute is evaluated against the "Deployed Composite Infoset" as described in Appendix A: Defining the Deployed Composites Infoset. See the section on "Attaching Intents and PolicySets to SCA Constructs" for more details on how this attribute is used.

- @provides (0..1) - a list of intent QNames (that can be qualified), which declares the intents the PolicySet provides.

PolicySet contains one or more of the ~~following~~ element children

- intentMap element
- policySetReference element
- xs:any extensibility element

Any mix of the above types of elements, in any number, can be included as children of the policySet element including extensibility elements. There are likely to be many different policy languages for specific binding technologies and domains. In order to allow the inclusion of any policy language within a policySet, the extensibility elements can be from any namespace and can be intermixed.

The SCA policy framework expects that WS-Policy will be a common policy language for expressing interaction policies, especially for Web Service bindings. Thus a common usecase is to attach WS-Policies directly as children of <policySet> elements; either directly as <wsp:Policy> elements, or as <wsp:PolicyReference> elements or using <wsp:PolicyAttachment>. These three elements, and others, can be attached using the extensibility point provided by the <xs:any> in the pseudo schema above. See example below.

For example, the policySet element below declares that it provides **serverAuthentication.message** and **reliability** for the "binding.ws" SCA binding.

```
<policySet name="SecureReliablePolicy"
      provides="serverAuthentication.message exactlyOne"
      appliesTo="//sca:binding.ws"
      xmlns="http://docs.oasis-open~~www.osoa~~.org/ns/opencsa~~xmlns~~/sca/200912~~1.0~~"
      xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
   <wsp:PolicyAttachment>
      <!-- policy expression and policy subject for
           "basic server authentication" -->
      …
   </wsp:PolicyAttachment>
   <wsp:PolicyAttachment>
   <!-- policy expression and policy subject for
       "reliability" -->
      …
   </wsp:PolicyAttachment>
</policySet>
```

Snippet 3-5: Example policySet Defineition

PolicySet authors need to be aware of the evaluation of the @appliesTo attribute in order to designate meaningful values for this attribute. Although policySets can be attached to any element in an SCA composite, the applicability of a policySet is not scoped by where it is attached in the SCA framework. Rather, policySets always apply to either binding instances or implementation elements regardless of where they are attached. In this regard, the SCA policy framework does not scope the applicability of the policySet to a specific attachment point in contrast to other frameworks, such as WS-Policy.

When computing the policySets that apply to a particular element, the @appliesTo attribute of each relevant policySet is checked against the element. If a policySet that is attached to an ancestor element does not apply to the element in question, it is simply discarded.

With this design principle in mind, an XPath expression that is the value of an @appliesTo attribute designates what a policySet applies to. Note that the XPath expression will always be evaluated ~~within the context of an attachment considering elements where binding instances or implementations are allowed to be present. The expression is evaluated~~ against the Domain Composite Infoset as described in Section 4.4.1 "The Form of the @attachTo Attribute"~~*parent element of any binding or implementation element*~~. The policySet will apply to any child binding or implementation elements returned from the expression. So, for example, appliesTo="//binding.ws" will match any web service binding. If appliesTo="//binding.ws[@impl='axis']" then the policySet would apply only to web service bindings that have an @impl attribute with a value of 'axis'.

When writing policySets, the author needs to ensure that the policies contained in the policySet always satisfy the intents in the @provides attribute. Specifically, when using WS-Policy the optional attribute and the exactlyOne operator can result in alternative policies and uncertainty as to whether a particular alternative satisfies the advertised intents.

If the WS-Policy attribute optional = 'true' is attached to a policy assertion, it results in two policy alternatives, one that includes and one that does not include the assertion. During wire validation it is impossible to predict which of the two alternatives will be selected - if the absence of the policy assertion does not satisfy the intent, then it is possible that the intent is not actually satisfied when the policySet is used.

Similarly, if the WS-Policy operator exactlyOne is used, only one of the set of policy assertions within the operator is actually used at runtime. If the set of assertions is intended to satisfy one or more intents, it is vital to ensure that each policy assertion in the set actually satisfies the intent(s).

Note that section 0 on Wire Validity specifies that the strict version of the WS-Policy intersection algorithm is used to establish wire validity and determine the policies to be used. The strict version of policy intersection algorithm ignores the ignorable attribute on assertions. This means that the ignorable facility of WS-Policy cannot be used in policySets.

For further discussion on attachment of policySets and the computation of applicable policySets, please refer to Section 4.

A policySet can be contributed to

~~All~~ the ~~policySets in a~~ SCA Domain by including its definition in a ~~are defined in a global, domain-wide file named~~ definitions.xml file within a Contribution in the Domain.~~-~~ Details of the definitions.xml files~~this file~~ are described in the SCA Assembly Model [SCA-Assembly].

3.4.1 IntentMaps

Intent maps contain the concrete policies and policy subjects that are used to realize a specific intent that is provided by the policySet.

The pseudo-schema for intentMaps is given in Snippet 3-6~~below~~:

```
<intentMap provides="xs:QName"~~>~~
          ~~>~~
<qualifier name="xs:string">?
      <xs:any>*
              ~~<intentMap/> ?~~
</qualifier>
</intentMap>
```

Snippet 3-6: intentMap Pseudo-Schema

When a policySet element contains a set of intentMap children, the value of the @provides attribute of each intentMap MUST correspond to an unqualified intent that is listed within the @provides attribute value of the parent policySet element. [POL30008]

If a policySet specifies a qualifiable intent in the @provides attribute, and it provides an intentMap for the qualifiable intent then that intentMap MUST specify all possible qualifiers for that intent. [~~POL30008~~POL30020]

If a policySet specifies a qualifiable intent in the @provides attribute, and it provides an intentMap for the qualifiable intent then that intentMap MUST specify all possible qualifiers for that intent. [POL30020]

For each qualifiable intent listed as a member of the @provides attribute list of a policySet element, there MUST be no more than one corresponding intentMap element that declares the unqualified form of that intent in its @provides attribute. In other words, each intentMap within a given policySet uniquely provides for a specific intent. [POL30010]

The @provides attribute value of each intentMap that is an immediate child of a policySet MUST be included in the @provides attribute of the parent policySet. [POL30021]

An intentMap element contains qualifier element children. Each qualifier element corresponds to a qualified intent where the unqualified form of that intent is the value of the @provides attribute value of the parent intentMap. The qualified intent is either included explicitly in the value of the enclosing policySet's @provides attribute or implicitly by that @provides attribute including the unqualified form of the intent. One of the qualifiers referenced in an intentMap MUST be the default qualifier defined for the qualifiable intent. [POL30022]

A qualifier element designates a set of concrete policy attachments that correspond to a qualified intent. The concrete policy attachments can be specified using wsp:PolicyAttachment element children or using extensibility elements specific to an environment.

As an example, the policySet element in Snippet 3-7~~below~~ declares that it provides **confidentiality** using the @provides attribute. The alternatives (transport and message) it contains each specify the policy and policy subject they provide. The default is “transport”.

```
<policySet name="SecureMessagingPolicies"
      provides="confidentiality"
      appliesTo="//binding.ws"
      xmlns="http://docs.oasis-open~~www.osoa~~.org/ns/opencsa~~xmlns~~/sca/200912~~1.0~~"
      xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
   <intentMap provides="confidentiality" >
      <qualifier name="transport">
         <wsp:PolicyAttachment>
             <!-- policy expression and policy subject for
                  "transport" alternative -->
                ...
                                             ...
</wsp:PolicyAttachment>
         <wsp:PolicyAttachment>
               ...
                                             ...
</wsp:PolicyAttachment>
      </qualifier>
      <qualifier name="message">
         <wsp:PolicyAttachment>
            <!-- policy expression and policy subject for
                 "message" alternative” -->
               ...
                                             ...
</wsp:PolicyAttachment>
      </qualifier>
   </intentMap>
</policySet>
```

Snippet 3-7: Example policySet with an intentMap

PolicySets can embed policies that are defined in any policy language. Although WS-Policy is the most common language for expressing interaction policies, it is possible to use other policy languagesSnippet 3-8~~. The following~~ is an example of a policySet that embeds a policy defined in a proprietary language. This policy provides “serverAuthentication” for binding.ws.

```
<policySet name="AuthenticationPolicy"
      provides="serverAuthentication"
      appliesTo="//binding.ws"
      xmlns="http://docs.oasis-
open~~www.osoa~~.org/ns/opencsa~~xmlns~~/sca/200912~~1.0~~">
   <e:policyConfiguration xmlns:e=”http://example.com”>
      <e:authentication type = “X509”/>
         <e:trustedCAStore type=”JKS”/>
         <e:keyStoreFile>Foo.jks</e:keyStoreFile>
         <e:keyStorePassword>123</e:keyStorePassword>
      </e:authentication>
   </e:policyConfiguration>
</policySet>
```

Snippet 3-8: Example policySet Using a Proprietary Language

~~The following example illustrates an intent map that defines policies for an intent with more than one level of qualification.~~

```
~~<policySet name="SecurityPolicy" provides="confidentiality">~~
      ~~<intentMap provides="confidentiality" >~~
            ~~<qualifier name="message">~~
                  ~~<intentMap provides="message" >~~
                        ~~<qualifier name="body">~~
                              ~~<!--- policy attachment for body encryption →~~
                        ~~</qualifier>~~
                        ~~<qualifier name="whole">~~
                              ~~<!--- policy attachment for whole message~~
                              ~~→encryption~~
                        ~~</qualifier>~~
                  ~~</intentMap>~~
            ~~</qualifier>~~
            ~~<qualifier name="transport">~~
                              ~~<!--- policy attachment for transport~~
                              ~~encryption →~~
            ~~</qualifier>~~
      ~~</intentMap>~~
~~</policySet>~~
```

3.4.2 Direct Inclusion of Policies within PolicySets

In cases where there is no need for defaults or overriding for an intent included in the @provides of a policySet, the policySet element can contain policies or policy attachment elements directly without the use of intentMaps or policy set references. There are two ways of including policies directly within a policySet. Either the policySet contains one or more wsp:policyAttachment elements directly as children or it contains extension elements (using xs:any) that contain concrete policies.

Following the inclusion of all policySet references, when a policySet element directly contains wsp:policyAttachment children or policies using extension elements, the set of policies specified as children MUST satisfy all the intents expressed using the @provides attribute value of the policySet element. [POL30011] The intent names in the @provides attribute of the policySet can include names of profile intents.

3.4.3 Policy Set References

A policySet can refer to other policySets by using sca:PolicySetReference element. This provides a recursive inclusion capability for intentMaps, policy attachments or other specific mappings from different domains.

When a policySet element contains policySetReference element children, the @name attribute of a policySetReference element designates a policySet defined with the same value for its @name attribute. Therefore, the @name attribute is a QName.

The set of intents in the @provides attribute of a referenced policySet MUST be a subset of the set of intents in the @provides attribute of the referencing policySet. [POL30013] Qualified intents are a subset of their parent qualifiable intent.

The usage of a policySetReference element indicates a copy of the element content children of the policySet that is being referred is included within the referring policySet. If the result of inclusion results in a reference to another policySet, the inclusion step is repeated until the contents of a policySet does not contain any references to other policySets.

When a policySet is applied to a particular element, the policies in the policy set include any standalone polices plus the policies from each intent map contained in the PolicySet, as described below.

Note that, since the attributes of a referenced policySet are effectively removed/ignored by this process, it is the responsibility of the author of the referring policySet to include any necessary intents in the @provides attribute of the policySet making the reference so that the policySet correctly advertises its aggregate policy.

The default values when using this aggregate policySet come from the defaults in the included policySets. A single intent (or all qualified intents that comprise an intent) in a referencing policySet ought to be included once by using references to other policySets.

Snippet 3-9 ~~Here~~ is an example to illustrate the inclusion of two other policySets in a policySet element:

```
<policySet name="BasicAuthMsgProtSecurity"
      provides="serverAuthentication confidentiality"
      appliesTo="//binding.ws"
      xmlns="http://docs.oasis-
open~~www.osoa~~.org/ns/opencsa~~xmlns~~/sca/200912~~1.0~~">
   <policySetReference name="acme:ServerAuthenticationPolicies"/>
   <policySetReference name="acme:ConfidentialityPolicies"/>
</policySet>
```

Snippet 3-9: Example *~~The above~~ policySet Including Other policySets*

The policySet in Snippet 3-9 refers to policySets for **serverAuthentication** and **confidentiality** and, by reference, provides policies and policy subject alternatives in these domains.

If the policySets referred to in Snippet 3-9 have the following content:

```
<policySet name="ServerAuthenticationPolicies"
      provides="serverAuthentication"
      appliesTo="//binding.ws"
      xmlns="http://docs.oasis-
open~~www.osoa~~.org/ns/opencsa~~xmlns~~/sca/200912~~1.0~~">
   <wsp:PolicyAttachment>
      <!-- policy expression and policy subject for
                "basic server
authentication" -->
      …
                …
</wsp:PolicyAttachment>
</policySet>

<policySet name="acme:ConfidentialityPolicies"
      provides="confidentiality"
      bindings="binding.ws"
      xmlns="http://docs.oasis-
open~~www.osoa~~.org/ns/opencsa~~xmlns~~/sca/200912~~1.0~~">
```

```
   <intentMap provides="confidentiality" >
      <qualifier name="transport">
         <wsp:PolicyAttachment>
            <!-- policy expression and policy subject for
                "transport"
alternative -->
            ...
               ...
</wsp:PolicyAttachment>
         <wsp:PolicyAttachment>
            ...
               ...
</wsp:PolicyAttachment>
      </qualifier>
      <qualifier name="message">
         <wsp:PolicyAttachment>
            <!-- policy expression and policy subject for
                "message"
alternative" -->
            ...
               ...
</wsp:PolicyAttachment>
      </qualifier>
   </intentMap>
</policySet>
```

Snippet 3-10: Example Included policySets for Snippet 3-9

The result of the inclusion of policySets via policySetReferences would be semantically equivalent to Snippet 3-11.~~the following:~~

```
<policySet name="BasicAuthMsgProtSecurity"
      provides="serverAuthentication confidentiality"  appliesTo="//binding.ws"
       xmlns="http://docs.oasis-
open~~www.osoa~~.org/ns/opencsa~~xmlns~~/sca/200912~~1.0~~">
   <wsp:PolicyAttachment>
      <!-- policy expression and policy subject for
           "basic server authentication" -->
         ...
               ...
</wsp:PolicyAttachment>
   <intentMap provides="confidentiality" >
      <qualifier name="transport">
         <wsp:PolicyAttachment>
            <!-- policy expression and policy subject for
                "transport"
alternative -->
            ...
               ...
</wsp:PolicyAttachment>
         <wsp:PolicyAttachment>
            ...
               ...
</wsp:PolicyAttachment>
      </qualifier>
      <qualifier name="message">
         <wsp:PolicyAttachment>
            <!-- policy expression and policy subject for
                "message"
alternative -->
```

```
            ...
                    ...
</wsp:PolicyAttachment>
      </qualifier>
   </intentMap>
</policySet>
```

Snippet 3-11: Equivalent policySet

4 Attaching Intents and PolicySets to SCA Constructs

This section describes how intents and policySets are associated with SCA constructs. It describes the various attachment points and semantics for intents and policySets and their relationship to other SCA elements and how intents relate to policySets in these contexts.

4.1 Attachment Rules – ~~-~~ Intents

One or more intents~~Intents~~ can be attached to any SCA element used in the definition of components and composites~~ since an intent specifies an abstract requirement~~. The attachment can be ~~is~~ specified by using the following two mechanisms:

- ***Direct Attachment*** mechanism which is described in the section "Direct Attachment of Intents".
- ***External Attachment*** mechanism which is described in the section "External Attachment of Intents".

4.2 Direct Attachment of Intents

Intents can be attached to any SCA element used in the definition of components and composites. Intents are attached by using the ***@requires*** attribute or the <requires> child element. The @requires ~~- This~~ attribute takes as its value a list of intent names. Similarly, the <requires> element takes as its value a list of intent names. Intents can also be attached~~applied~~ to interface definitions. For WSDL port~~Port~~ Type elements (WSDL 1.1) ~~and for WSDL Interface elements (WSDL 2.0),~~ the @requires attribute can be used to attach the list of intents ~~applied that holds a list of intent names~~ that are needed by the interface. ~~.~~ Other interface languages can define their own mechanism for attaching~~specifying~~ a list of intents. Any intents attached to an interface definition artifact, such as a WSDL portType, MUST be added to the intents attached to the service or reference to which the interface definition applies. If no intents are attached to the service or reference then the intents attached to the interface definition artifact become the only intents attached to the service or reference. [POL40027]~~. Any service or reference that uses an interface which has intents attached to it implicitly adds those intents to its own @requires list.~~

Because intents specified on interfaces can be seen by both the provider and the client of a service, it is appropriate to use them to specify characteristics of the service that both the developers of provider and the client need to know.

For example:

```
<service requires="acme:IntentName1 acme:IntentName2">
   <binding.xxx/>
   …
</service>

<reference requires="acme:IntentName1 acme:IntentName2">
   <binding.xxx/>
   …
</reference>
```

Snippet 4-1: Example of @requires on a service or a reference

```
<service>
   <requires intents="acme:IntentName1 acme:IntentName2"/>
   <binding.xxx/>
   …
</service>

<reference>
```

```
    <requires intents="acme:IntentName1 acme:IntentName2"/>
    <binding.xxx/>
    …
</reference>
```

Snippet 4-2: Example of a <requires> subelement to attach intents to a service or a reference

4.3 External Attachment of Intents and PolicySets

External Attachment of intents and policySets is used for deployment-time application of intents and policySets to SCA elements. It is called "external attachment" because the principle of the mechanism is that the attachment is declared in a place that is separate from the composite files that contain the elements. This separation provides the deployer with a way to attach intents and/or policySets without having to modify the artifacts where the intents and policySets are attached.

Intents and policySets can be attached to one or more SCA elements by using the externalAttachment element, which is declared within a definitions file.

The pseudo-schema for the externalAttachment element is shown in Snippet 4-3.

```
<externalAttachment intents="sca:listOfQNames"
                    policySets="sca:listofQNames"
                    attachTo = "xs:string" />
```

Snippet 4-3: Pseudo-schema for externalAttachment element

The ***externalAttachment*** element has the ***attributes***:

@intents : listOfQNames (0..1) A list of QNames identifying intents which are attached to the elements declared in the @attachTo attribute.

@policySets : listOfQNames (0..1). A list of QNames identifying policySets which are attached to the elements declared in the @attachTo attribute

@attachTo : string (1..1). A string containing an XPath 1.0 expression identifying one or more elements in the Domain. It is used to declare which set of elements the intents are attached to. The contents of the @attachTo attribute of an externalAttachment element MUST match the XPath 1.0 production Expr. [POL40035] The XPath value of the @attachTo attribute is evaluated against the "Deployed Composite Infoset" as described in the appendix section "The Deployed Composites Infoset".

~~For example:~~

~~<service> or <reference>…~~
~~<binding.binding-type requires="listOfQNames"~~
~~</binding.binding-type>…~~
~~</service> or </reference>~~

~~4.2~~4.4 Attachment Rules - PolicySets

One or more policySets can be attached to any SCA element used in the definition of components and composites. The attachment can be specified by using the following two mechanisms:

- ***Direct Attachment*** mechanism which is described in Direct Attachment of PolicySetsSection 4.5.
- ***External Attachment*** mechanism which is described in External Attachment of PolicySets.Section 4.6.

SCA runtimes MUST support at least one of the Direct Attachment and External Attachment mechanisms for policySet attachment. [POL40010] SCA implementations supporting only the External Attachment mechanism MUST ignore the policy sets that are applicable via the Direct Attachment mechanism. [POL40011] SCA implementations supporting only the Direct Attachment mechanism MUST ignore the policy sets that are applicable via the External Attachment mechanism. [POL40012] The SCA runtime MUST raise an error if the @attachTo XPath expression resolves to an SCA <property> element, or any of its children. [POL40001]

~~4.3~~4.5 Direct Attachment of PolicySets

Direct Attachment of PolicySets can be achieved by

~~1.~~ Using the optional ***@policySets*** attribute of the SCA element

~~2.~~ Adding an optional child **<policySetAttachment/>** element to the SCA element

The policySets attribute takes as its value a list of policySet names.

For example:

```
<service> or <reference>…
   <binding.binding-type policySets="listOfQNames">
   </binding.binding-type>
   …
</service> or </reference>
```

Snippet 4-4: Example of @policySets on a service

The <policySetAttachment/> element is an alternative way to attach a policySet to an SCA composite.

```
<policySetAttachment name="xs:QName"/>
```

Snippet 4-5: policySetAttachment Pseudo-Schema

- @name (1..1) – the QName of a policySet.

For example:

```
<service> or <reference>…
   <binding.binding-type>
         <policySetAttachment name="sns:EnterprisePolicySet">
   </binding.binding-type>
   …
</service> or </reference>
```

Snippet 4-6:Example of policySetAttachment in a service or reference

Where an element has both a @policySets attribute and a <policySetAttachment/> child element, the policySets declared by both are attached to the element.

The SCA Policy framework enables two distinct cases for utilizing intents and PolicySets:

~~1.~~ It is possible to specify QoS requirements by attaching~~specifying~~ abstract intents to~~utilizing the @requires element on~~ an element at the time of development. In this case, it is implied that the concrete bindings and policies that satisfy the abstract intents are not assigned at development time but the intents are used ***to select the concrete Bindings and Policies*** at deployment time. Concrete policies are encapsulated within policySets that are applied during deployment using the external attachment mechanism. The intents associated with a SCA element is the union of intents specified for it and its parent elements subject to the detailed rules below.

~~2.~~ It is also possible to specify QoS requirements for an element by using both intents and concrete policies contained in directly attached policySets at development time. In this case, it is possible ***to configure the policySets, by overriding the default settings in the specified policySets using intents***. The policySets associated with a SCA element is the union of policySets specified for it and its parent elements subject to the detailed rules below.

See also "~~section~~ Matching Intents and PolicySets" for a discussion of how intents are used to guide the selection and application of specific policySets.

~~4.4~~4.6 External Attachment of PolicySets ~~Mechanism~~

The External Attachment ~~mechanism~~ for policySets is used for deployment-time application of policySets and policies to SCA elements. It is called "external attachment" because the principle of the mechanism is that the place that declares the attachment is separate from the composite files that contain the elements. This separation provides the deployer with a way to attach policies and policySets without having to modify the artifacts where they apply.

A PolicySet is attached to one or more elements in one of two ways:

a) through the @attachTo attribute of the policySet

b) through the @attachTo attribute of an <externalAttachment/> element which references the policySet in its @policySets attribute

c) through a reference (via policySetReference) from a policySet that uses the @attachTo attribute.

4.6.1 Cases Where Multiple PolicySets are attached to a Single Artifact

Multiple PolicySets can be attached to a single artifact. This can happen either as the result of one or more direct attachments or as the result of one or more external attachments which target the particular artifact.

4.7 Attaching intents to SCA elements

A list of intents can be attached to any SCA element by using the @requires attribute or the <requires> subelement.

The intents which apply to a given element depend on

the intents expressed in its @requires attribute and/or its <requires> subelement

intents derived from the structural hierarchy of the element

intents derived from the implementation hierarchy of the element

When computing the intents that apply to a particular element, the @constrains attribute of each relevant intent is checked against the element. If the intent in question does not apply to that element it is simply discarded.

The intents declared on elements lower in the implementation hierarchy of a given element MUST be applied to the element. [~~POL40009~~POL40013]~~. Specific examples are discussed later in this document.~~

4.7.1 Implementation Hierarchy of an Element

The ***implementation hierarchy*** occurs where a component configures an implementation and also where a composite promotes a service or reference of one of its components. The implementation hierarchy involves:

a composite service or composite reference element is in the implementation hierarchy of the component service/component reference element which they promote

the component element and its descendent elements (for example, service, reference, implementation) configure aspects of the implementation. Each of these elements is in the implementation hierarchy of the ***corresponding*** element in the componentType of the implementation.

Rule 1:

- If the process of redeployment of intents, externalAttachments and/or policySets fails because one or more intents are left unsatisfied, an error MUST be raised.

[~~POL40014~~POL40026] A qualifiable intent expressed lower in the hierarchy can be qualified further up the hierarchy, in which case the qualified version of the intent MUST apply to the higher level element. [POL40004]

4.7.2 Structural Hierarchy of an Element

The structural hierarchy of an element consists of its parent element, grandparent element and so on up to the <composite/> element in the composite file containing the element.

As an example, for the composite in Snippet 4-7::

```
<composite name="C1" requires="i1">
   <service name="CS" promotes="X/S">
      <binding.ws requires="i2">
   </service>
   <component name="X">
       <implementation.java class="foo"/>
       <service name="S" requires="i3">
   </component>
</composite>
```

Snippet 4-7: Example Composite to Illustrate Structural Hierarchy

- the structural hierarchy of the component service element with the name "S" is the component element named "X" and the composite element named "C1". Service "S" has intent "i3" and also has the intent "i1" if i1 is not mutually exclusive with i3.

Rule2: The intents declared on elements higher in the structural hierarchy of a given element MUST be applied to the element EXCEPT

~~1.~~ if any of the inherited intents is mutually exclusive with an intent applied on the element, then the inherited intent MUST be ignored

if the overall set of intents from the element itself and from its structural hierarchy contains both an unqualified version and a qualified version of the same intent, the qualified version of the intent MUST be used..

[POL40005]

4.7.3 Combining Implementation and Structural Policy Data

When there are intents present in both hierarchies implementation intents are calculated before the structural intents. In other words, when combining implementation hierarchy and structural hierarchy policy data, Rule 1 MUST be applied BEFORE Rule 2. [POL40015]

Note that each of the elements in the hierarchy below a <component> element, such as <service/>, <reference/> or <binding/>, inherits intents from the equivalent elements in the componentType of the implementation used by the component. So the <service/> element of the <component> inherits any intents on the <service/> element with the same name in the <componentType> - and a <binding/> element under the service in the component inherits any intents on the <binding/> element of the service (with the same name) in the componentType. Errors caused by mutually exclusive intents appearing on corresponding elements in the component and on the componentType only occur when those elements match one-to-one. Mutually exclusive intents can validly occur on elements that are at different levels in the structural hierarchy (as defined in Rule 2).

Note that it might often be the case that <binding/> elements will be specified in the structure under the <component/> element in the composite file (especially at the Domain level, where final deployment configuration is applied) - these elements might have no corresponding elements defined in the componentType structure. In this situation, the <binding/> elements don't acquire any intents from the componentType directly (ie there are no elements in the implementation hierarchy of the <binding/> elements), but those <binding/> elements will acquire intents "flowing down" their structural hierarchy as defined in Rule 2 - so, for example if the <service/> element is marked with @requires="confidentiality", the bindings of that service will all inherit that intent, assuming that they don't have their own exclusive intents specified.

Also, for example, where say a component <service.../> element has an intent that is mutually exclusive with an intent in the componentType<service.../> element with the same name, it is an error, but this differs when compared with the case of the <component.../> element having an intent that is mutually exclusive with an intent on the componentType <service/> element - because they are at different structural levels: the intent on the <component/> is ignored for that <service/> element and there is no error.

4.7.4 Examples

As an example, consider the composite in Snippet 4-8:

```
<composite name="C1" requires="i1">
   <service name="CS" promotes="X/S">
      <binding.ws requires="i2">
   </service>
   <component name="X">
       <implementation.java class="foo"/>
       <service name="S" requires="i3">
   </component>
</composite>
```

Snippet 4-8:Example composite with intents

…the component service with name "S" has the service named "S" in the componentType of the implementation in its implementation hierarchy, and the composite service named "CS" has the component service named "S" in its implementation hierarchy. Service "CS" acquires the intent "i3" from service "S" – and also gets the intent "i1" from its containing composite "C1" IF i1 is not mutually exclusive with i3.

When intents apply to an element following the rules described and where no policySets are attached to the element, the intents for the element can be used to select appropriate policySets during deployment, using the external attachment mechanism.

Consider the composite in Snippet 4-9:

```
<composite requires="confidentiality">
   <service name="foo" …/>
   <reference name="bar" requires="confidentiality.message"/>
</composite>
```

Snippet 4-9: Example reference with intents

…in this case, the composite declares that all of its services and references guarantee confidentiality in their communication, but the “bar” reference further qualifies that requirement to specifically require message-level security. The “foo” service element has the default qualifier specified for the confidentiality intent (which might be transport level security) while the “bar” reference has the **confidentiality.message** intent.

Consider the variation in Snippet 4-10 where a qualified intent is specified at the composite level:

```
<composite requires="confidentiality.transport">
   <service name="foo" …/>
   <reference name="bar" requires="confidentiality.message"/>
</composite>
```

Snippet 4-10: Example Qualified intents

In this case, both the **confidentiality.transport** *and* the **confidentiality.message** intent are applied for the reference ‘bar’. If there are no bindings that support this combination, an error will be generated. However, since in some cases multiple qualifiers for the same intent can be valid or there might be bindings that support such combinations, the SCA specification allows this.

4.8 Usage of Intent and Policy Set Attachment together

As indicated above, it is possible to attach both intents and policySets to an SCA element during development. The most common use cases for attaching both intents and concrete policySets to an element are with binding and reference elements.

When the @requires attribute or the <requires> subelement and one or both of the direct policySet attachment mechanisms are used together during development, it indicates the intention of the developer to configure the element, such as a binding, by the application of specific policySet(s) to this element.

The same behavior can be enabled by external attachment of intents and policySets.

Developers who attach intents and policySets in conjunction with each other need to be aware of the implications of how the policySets are selected and how the intents are utilized to select specific intentMaps, override defaults, etc. The details are provided in the Section Guided Selection of PolicySets using Intents.

4.9 Intents and PolicySets on Implementations and Component Types

It is possible to specify intents and policySets within a component’s implementation, which get exposed to SCA through the corresponding *component type*. How the intents or policies are specified within an implementation depends on the implementation technology. For example, Java can use an @requires annotation to specify intents.

The intents and policySets specified within an implementation can be found on the

<sca:implementation.*> and the <sca:service> and <sca:reference> elements of the component type. Snippet 4-11 shows direct attachment of intents and policySets using the @requires and @policySets attributes:

```
<omponentType>
```

```
   <implementation.* requires="listOfQNames" policySets="="listOfQNames">
      ...
   </implementation>
   <service name="myService" requires="listOfQNames"
      policySets="listOfQNames">
      ...
   </service>
   <reference name="myReference" requires="listOfQNames"
      policySets="="listOfQNames">
      ...
   </reference>
   …
</componentType>
```

Snippet 4-11: Example of intents on an implementation

Intents expressed in the component type are handled according to the rule defined for the implementation hierarchy. See Intent rule 2

For explicitly listed policySets, the list in the component using the implementation can override policySets from the component type. If a component has any policySets attached to it (by any means), then any policySets attached to the componentType MUST be ignored. [POL40006]

4.10 Intents on Interfaces

Interfaces are used in association with SCA services and references. These interfaces can be declared in SCA composite files and also in SCA componentType files. The interfaces can be defined using a number of different interface definition languages which include WSDL, Java interfaces and C++ header files.

It is possible for some interfaces to be referenced from an implementation rather than directly from any SCA files. An example of this usage is a Java implementation class file that has a reference declared that in turn uses a Java interface defined separately. When this occurs, the interface definition is treated from an SCA perspective as part of the componentType of the implementation, logically being part of the declaration of the related service or reference element.

Both the declaration of interfaces in SCA and also the definitions of interfaces can carry policy-related information. In particular, both the declarations and the definitions can have either intents attached to them, or policySets attached to them - or both. For SCA declarations, the intents and policySets always apply to the whole of the interface (ie all operations and all messages within each operation). For interface definitions, intents and policySets can apply to the whole interface or they can apply only to specific operations within the interface or they can even apply only to specific messages within particular operations. (To see how this is done, refer to the places in the SCA specifications that deal with the relevant interface definition language)

This means, in effect, that there are 4 places which can hold policy related information for interfaces:

1. The interface definition file that is referenced from the component type.
2. The interface declaration for a service or reference in the component type
3. The interface definition file that is referenced from the component declaration in a composite
4. The interface declaration within a component

If the required intent set contains a mutually exclusive pair of intents the SCA runtime MUST reject the document containing the element and raise an error. [POL40016]

The QName of the bindingType MUST be unique amongst the set of bindingTypes in the SCA Domain. [POL40019]

4.11 BindingTypes and Related Intents

SCA Binding types implement particular communication mechanisms for connecting components together. See detailed discussion in the SCA Assembly Specification [SCA-Assembly]. Some binding types can realize intents inherently by virtue of the kind of protocol technology they implement (e.g. an SSL binding would natively support confidentiality). For these kinds of binding types, it might be the case that using that binding type, without any additional configuration, provides a concrete realization of an intent. In addition, binding instances which are created by configuring a binding type might be able to provide some intents by virtue of their configuration. It is important to know, when selecting a binding to satisfy a set of intents, just what the binding types themselves can provide and what they can be configured to provide.

The bindingType element is used to declare a class of binding available in a SCA Domain. The pseudo-schema for the bindingType element is shown in Snippet 4-12:

```
<bindingType type="NCName"
   alwaysProvides="listOfQNames"?
   mayProvide="listOfQNames"?/>
```

Snippet 4-12: bindingTypePseudo-Schema

@type (1..1) – declares the NCName of the bindingType, which is used to form the QName of the bindingType. The QName of the bindingType MUST be unique amongst the set of bindingTypes in the SCA Domain. [POL40020]

@alwaysProvides (0..1) – a list of intent QNames that are natively provided. A natively provided intent is hard-coded into the binding implementation. The function represented by the intent cannot be turned off.

@mayProvides (0..1) – a list of intent QNames that are natively provided by the binding implementation, but which are activated only when present in the intent set that is applied to a binding instance.

A binding implementation MUST implement all the intents listed in the @alwaysProvides and @mayProvides attributes. [POL40021]

The kind of intents a given binding might be capable of providing, beyond these inherent intents, are implied by the presence of policySets that declare the given binding in their @appliesTo attribute.

For example, if the policySet in Snippet 4-13 is available in a SCA Domain it says that the (example) foo:binding.ssl can provide “reliability” in addition to any other intents it might provide inherently.

```
<policySet name="ReliableSSL" provides="exactlyOnce"
      appliesTo="//foo:binding.ssl">
      ...
</policySet>
```

Snippet 4-13:Example policySet Applied to a binding

4.12 Treatment of Components with Internal Wiring

This section discusses the steps involved in the development and deployment of a component and its relationship to selection of bindings and policies for wiring services and references.

The SCA developer starts by defining a component. Typically, this contains services and references. It can also have intents attached at various locations within composite and component types as well as policySets attached at various locations.

Both for ease of development as well as for deployment, the wiring constraints to relate services and references need to be determined. This is accomplished by matching constraints of the services and references to those of corresponding references and services in other components.

In this process, the intents, and the policySets that apply to both sides of a wire play an important role. In addition, concrete policies need to be selected that satisfy the intents for the service and the reference and are also compatible with each other. For services and references that make use of bidirectional interfaces, the same determination of matching policySets also has to take place for callbacks.

Determining wire compatibility plays an important role prior to deployment as well as during the deployment phases of a component. For example, during development, it helps a developer to determine whether it is possible to wire services and references using the policySets available in the development environment. During deployment, the wiring constraints determine whether wiring can be achievable. It also aids in adding additional concrete policies or making adjustments to concrete policies in order to deliver the constraints. Here are the concepts that are needed in making wiring decisions:

The set of intents that individually apply to *each* service or reference.

When possible the intents that are applied to the service, the reference and callback (if any) at the other end of the wire. This set is called the *required intent set* and only applies when dealing with a wire connecting two components within the same SCA Domain. When external connections are involved, from clients or to services that are outside the SCA domain, intents are only available for the end of the connection that is inside the domain. See Section "Preparing Services and References for External Connection" for more details.

The policySets that apply to each service or reference.

The set of provided intents for a binding instance is the union of the set of intents listed in the “alwaysProvides” attribute and the set of intents listed in the “mayProvides” attribute of of its binding type. The capabilities represented by the "alwaysProvides" intent set are always present, irrespective of the configuration of the binding instance. Each capability represented by the "mayProvides" intent set is only present when the list of intents applied to the binding instance (either applied directly, or inherited) contains the particular intent (or a qualified version of that intent, if the intent set contains an unqualified form of a qualifiable intent). When an intent is directly provided by the binding type, there is no need to apply a policy set that provides that intent.

When bidirectional interfaces are in use, the same process of selecting policySets to provide the intents is also performed for the callback bindings.

4.12.1 Determining Wire Validity and Configuration

The above approach determines the policySets that are used in conjunction with the binding instances listed for services and references. For services and references that are resolved using SCA wires, the policySets chosen on each side of the wire might or might not be compatible. The following approach is used to determine whether they are compatible and whether the wire is valid. If the wire uses a bidirectional interface, then the following technique ensures that valid configured policySets can be found for both directions of the bidirectional interface.

The policySets at each end of a wire using the compatibility rules of the policy languages. [POL40022] The policySets at each end of a wire MUST be incompatible if they use different policy languages. [POL40023] However, there is a special case worth mentioning:

If both sides of the wire use identical policySets (by referring to the same policySet by its QName in both sides of the wire), then they are compatible.

Where the policy language in use for a wire is WS-Policy, strict WS-Policy intersection MUST be used to determine policy compatibility. [POL40024]

Any intents attached to an interface definition artifact, such as a WSDL portType, MUST be added to the intents attached to the service or reference to which the interface definition applies. If no intents are attached to the service or reference then the intents attached to the interface definition artifact become the only intents attached to the service or reference. [POL40025]

4.13 Preparing Services and References for External Connection

Services and references are sometimes not intended for SCA wiring, but for communication with software that is outside of the SCA domain. References can contain bindings that specify the endpoint address of a service that exists outside of the current SCA domain. Services can specify bindings that can be exposed to clients that are outside of the SCA domain.

Matching service/reference policies across the SCA Domain boundary MUST use WS-Policy compatibility (strict WS-Policy intersection) if the policies are expressed in WS-Policy syntax. [POL40007] For other policy languages, the policy language defines the comparison semantics.

For external services and references that make use of bidirectional interfaces, the same determination of matching policies has to also take place for the callback.

The policies that apply to the service/reference are computed as discussed in Guided Selection of PolicySets using Intents.

4.14 Deployment

The SCA Assembly Specification [SCA-Assembly] describes how to contribute SCA artifacts to the SCA Domain, and how to deploy them to create running components. This section discusses the Policy aspects of deployment: how intents, externalAttachments and policySets are contributed, how intents are satisfied by concrete policies in policySets and the process of redeployment when intents, externalAttachments or policySets change.

Intents, externalAttachments and policySets can be contributed to the Domain contained within contributions. These contributions might only contain policy artifacts or they might also contain composites and related artifacts. Intents and policySets can be attached to elements within a composite either by direct attachment (where an attribute or child element performs the attachment) or they can be attached through the external attachment mechanism.

When a composite is deployed, the intents which are attached to each element must be evaluated, both the directly attached intents and intents attached through external attachment. For external attachment, this means evaluating the @attachTo attribute of each externalAttachment element with a non-empty @intents attribute in the SCA Domain - the intents are attached to those elements which are selected by the XPath expression in the externalAttachment/@attachTo attribute.

During the deployment of SCA composites, first all <externalAttachment/> elements within the Domain MUST be evaluated to determine which intents are attached to elements in the newly deployed composite and then all policySets within the Domain with an @attachTo attribute or <externalAttachment> elements that attach policySets MUST be evaluated to determine which policySets are attached to elements in the newly deployed composite. [POL40034]

Once the intents attached to the elements of a composite are known, the policySets attached to each element are evaluated. If external attachment of policySets is supported, then each policySet in the Domain is examined and the XPath expression of the @attachTo attribute is evaluated and the policySet is attached to SCA elements selected by the expression.

The SCA runtime MUST raise an error if the @attachTo XPath expression resolves to an SCA <property> element, or any of its children.[POL40002]

The algorithm for matching intents with policySets is described in the section "Matching Intents and PolicySets".

4.14.1 Redeployment of Intents and PolicySets

Intents and policySets can be managed separately from other SCA artifacts. It is possible for an SCA runtime to allow deployment of new intents, new externalAttachments and policySets, modification of existing intents, externalAttachments and policySets or the undeployment of existing intents, externalAttachments and policySets, while composites and components are deployed or are running in the Domain. Collectively, this is referred to as ***the redeployment of intents and policySets***.

Redeployment can be caused by:

- Adding an externalAttachment element to the Domain
- Adding a policySet with a non-empty attachTo attribute to the Domain
- Changing the structure of an intent or policySet in the Domain that is directly or externally attached.
- Changing the attachTo, policySets or intents attribute of a externalAttachment in the Domain.
- Removing directly attached intents or policySets from the Domain.
- Removing one or more externalAttachment elements from the Domain.

Note that an SCA runtime can choose to disallow redeployment of intents and policySets.

If an SCA runtime supports the redeployment of intents and policySets, there is an implication that the changed intent and policySet artifacts can change the configuration of composites and components in the Domain. How the changes are implemented is determined by the design of the SCA runtime concerned, but there are three general approaches, as outlined in the SCA Assembly specification [SCA-Assembly]:

- the SCA runtime can require that all existing running component instances affected by the configuration changes are stopped and then restarted using the new configuration
- the SCA runtime can leave existing running component instances unchanged, but any new component instances are created using the new configuration
- the SCA runtime can deploy the new or changed intents, externalAttachments and policySets to the SCA Domain but not activate the changes until some time in the future. Running component instances and new component instances are not affected (i.e., the component configuration is not changed) by the newly deployed intents, externalAttachments and policySets until the SCA runtime activates those changes. The means and mechanism for performing this activation is outside the scope of this specification.

Redeployment of intents and policySets, when it occurs, first performs external attachment of intents followed by external attachment of policySets. After this, the algorithm for matching intents with policySets is executed. The redeployment process may succeed or it may fail, in that the set of intents attached to artifacts in the domain may or may not be satisfied. If the process of redeployment of intents, externalAttachments and/or policySets fails because one or more intents are left unsatisfied, an error MUST be raised. [POL40029] If the process of redeployment of intents, externalAttachments and/or policySets fails, the changed intents, externalAttachments and/or policySets MUST NOT be deployed and no change is made to deployed and running artifacts. [POL40030]

If the redeployment of intents, externalAttachments and policySets succeeds in that all intents are satisfied, then the policies attached to one or more deployed SCA elements may change. When redeployment of intents, externalAttachments and policySets succeeds, the components whose policies are affected by the redeployment MAY have their policies updated by the SCA runtime dynamically without the need to stop and restart those components. [POL40031]

Where components are updated by redeployment of intents, externalAttachments and policySets (their configuration is changed in some way, which includes changing the policies associated with a component), the new configuration MUST apply to all new instances of those components once the redeployment is complete. [POL40032] Where a component configuration is changed by the redeployment of intents, externalAttachments and policySets, the SCA runtime either MAY choose to maintain existing instances with the old configuration of the component, or the SCA runtime MAY choose to stop and discard existing instances of the component. [POL40033]

4.15 Matching Intents and PolicySets

This section describes the selection of concrete policies that provide the requirements expressed by the set of intents associated with an SCA element. The purpose is to construct the set of concrete policies that are attached to an element taking into account the explicitly declared policySets that are attached to an element as well as policySets that are externally attached. The aim is to satisfy all of the intents that apply to each element.

If the unqualified form of a qualifiable intent is attached to an element, it can be satisfied by a policySet that specifies any one of qualified forms of the intent in the value of its @provides attribute, or it can be satisfied by a policySet which @provides the unqualified form of the intent. If the qualified form of the intent is attached to an element then it can be satisfied only by a policy that @provides that qualified form of the intent.

Note: In the following, the following rule is observed when an intent set is computed.

When a profile intent is encountered in either a global @requires attribute, an intent/@requires attribute, a <requires> subelement or a policySet/@provides attribute, the profile intent is immediately replaced by the intents that it composes (i.e. all the intents that appear in the profile intent's @requires attribute). This rule is applied recursively until profile intents do not appear in an intent set. [This is stated generally here, in order to not have to restate this at multiple places].

The ***required intent set*** that is attached to an element is:

5. The set of intents attached to the element either by direct attachment or external attachment via the mechanisms described in the sections "Direct Attachment of Intents" and "External Attachment of Intents".
6. add any intents found in any related interface definition or declaration, as described in the section "Intents on Interfaces".
7. add any intents found on elements below the target element in its implementation hierarchy as defined in Rule 1 in the section "Implementation Hierarchy of an Element".
8. add any intents attached to each ancestor element in the element's structural hierarchy as defined in Rule 2 in in the section "Structural Hierarchy of an Element"
9. remove any intents that do not include the target element's type in their @constrains attribute.
10. remove the unqualified version of an intent if the set also contains a qualified version of that intent

All intents in the required intent set for an element MUST be provided by the directly provided intents set and the set of policySets that apply to the element, or else an error is raised. [POL40017]

The ***directly provided intent set*** for an element is the set of intents listed in the @alwaysProvides attribute combined with the set of intents listed in the @mayProvides attribute of the bindingType or implementationType declaration for a binding or implementation element respectively.

The ***set of PolicySets attached to an element*** include those ***explicitly specified*** using the @policySets attribute or the <policySetAttachment/> element and those which are ***externally attached***.

A policySet ***applies to*** a target element if the result of the XPath expression contained in the policySet's @appliesTo attribute, when evaluated against the document containing the target element, includes the target element. For example, @appliesTo="//binding.ws[@impl='axis']" matches any binding.ws element that has an @impl attribute value of 'axis'.

The set of ***explicitly specified*** policySets for an element is:

11. The union of the policySets specified in the element's @policySets attribute and those specified in any <policySetAttachment/> child element(s).
12. add the policySets declared in the @policySets attributes and <policySetAttachment/> elements from elements in the structural hierarchy of the element.
13. remove any policySet where the policySet does not apply to the target element.
    It is not an error for a policySet to be attached to an element to which it doesn't apply.

The set of ***externally attached*** policySets for an element is:

14. Each <PolicySet/> in the Domain where the element is targeted by the @attachTo attribute of the policySet
15. Each PolicySet that is attached to the target element through use of the <externalAttachment/> element

16. remove any policySet where the policySet does not apply to the target element.
It is not an error for a policySet to be attached to an element to which it doesn't apply.

A policySet ***provides an intent*** if any of the statements are true:

17. The intent is contained in the @provides list of the policySet.
18. The intent is a qualified intent and the unqualified form of the intent is contained in the @provides list of the policySet.
19. The policySet @provides list contains a qualified form of the intent (where the intent is qualifiable).

The locations where interfaces are defined and where interfaces are declared in the componentType and in a component MUST be treated as part of the implementation hierarchy as defined in Section 4.5 Attaching intents to SCA elements. [POL40018]

5 Implementation Policies

The basic model for Implementation Policies is very similar to the model for interaction policies described above. Abstract QoS requirements, in the form of intents, can be associated with SCA component implementations to indicate implementation policy requirements. These abstract capabilities are mapped to concrete policies via policySets at deployment time. Alternatively, policies can be associated directly with component implementations using policySets. Intents and policySets can be attached to an implementation using any of the mechanisms described in “Attaching Intents and PolicySets to SCA Constructs”.

Snippet 5-1 shows one way of associating intents with an implementation:

```
<component name="xs:NCName" … >
   <implementation.* … requires="listOfQNames">
      …
   </implementation>
   …
</component>
```

Snippet 5-1: Example of intents Associated with an implementation

If, for example, one of the intent names in the value of the @requires attribute is ‘logging’, this indicates that all messages to and from the component have to be logged. The technology used to implement the logging is unspecified. Specific technology is selected when the intent is mapped to a policySet (unless the implementation type has native support for the intent, as described in the next section). A list of implementation intents can also be specified by any ancestor element of the <sca:implementation> element. The effective list of implementation intents is the union of intents specified on the implementation element and all its ancestors.

In addition, one or more policySets can be specified directly by associating them with the implementation of a component.

```
<component name="xs:NCName" … >
<implementation.* … policySets="="listOfQNames">
      …
   </implementation>
      …
</component>
```

Snippet 5-2: Example of policySets Associated with an implemenation

Snippet 5-2 shows how intents and policySets can be specified on a component. It is also possible to specify intents and policySets within the implementation. How this is done is defined by the implementation type.

The intents and policy sets are specified on the <sca:implementation.*> element within the component type. This is important because intent and policy set definitions need to be able to specify that they constrain an appropriate implementation type.

```
<componentType>
   <implementation.* requires="listOfQNames" policySets="listOfQNames">
   …
   </implementation>
   …
</componentType>
```

Snippet 5-3: intents and policySets Constraining an implementation

When applying policies, the intents attached to the implementation are added to the intents attached to the using component. For the explicitly listed policySets, the list in the component can override policySets from the componentType.

Some implementation intents are targeted at <binding/> elements rather than at <implementation/> elements. This occurs in cases where there is a need to influence the operation of the binding implementation code rather than the code directly related to the implementation itself. Implementation elements of this kind will have a @constrains attribute pointing to a binding element, with a @intentType of "implementation".

5.1 Natively Supported Intents

Each implementation type (e.g. <sca:implementation.java> or <sca:implementation.bpel>) has an ***implementation type definition*** within the SCA Domain. An implementation type definition is declared using an implementationType element within a <definitions/> declaration. The pseudo-schema for the implementationType element is shown in Snippet 5-4:

```
<implementationType type="QName"
alwaysProvides="listOfQNames"? mayProvide="listOfQNames"? />
```

Snippet 5-4: implementationType Pseudo-Schema

The implementation Type element has the following attributes:

name : QName (1..1) - the name of the implementationType. If the process of redeployment of intents, externalAttachments and/or policySets fails, the changed intents, externalAttachments and/or policySets MUST NOT be deployed and no change is made to deployed and running artifacts. [POL50001] For example: "sca:implementation.java".

alwaysProvides : list of QNames (0..1) - a set of intents. The intents in the alwaysProvides set are always provided by this implementation type, whether the intents are attached to the using component or not.

mayProvide : list of QNames (0..1) - a set of intents. The intents in the mayProvide set are provided by this implementation type if the intent in question is attached to the using component.

5.2 Writing PolicySets for Implementation Policies

The @appliesTo and @attachTo attributes for a policySet take an XPath expression that is applied to a service, reference, binding or an implementation element. For implementation policies, in most cases, all that is needed is the QName of the implementation type. Implementation policies can be expressed using any policy language (which is to say, any configuration language). For example, XACML or EJB-style annotations can be used to declare authorization policies. Other capabilities could be configured using completely proprietary configuration formats.

For example, a policySet declared to turn on trace-level logging for a BPEL component could be declared as is Snippet 5-5:

```
<policySet name=”loggingPolicy” provides="acme:logging.trace"
      appliesTo="//sca:implementation.bpel" …>
   <acme:processLogging level="3"/>
</policySet>
```

Snippet 5-5: Example policySet Applied to implemenation.bpel

5.2.1 Non WS-Policy Examples

Authorization policies expressed in XACML could be used in the framework in two ways:

20. Embed XACML expressions directly in the PolicyAttachment element using the extensibility elements discussed above, or
21. Define WS-Policy assertions to wrap XACML expressions.

For EJB-style authorization policy, the same approach could be used:

22. Embed EJB-annotations in the PolicyAttachment element using the extensibility elements discussed above, or
23. Use the WS-Policy assertions defined as wrappers for EJB annotations.

6 Roles and Responsibilities

There are 4 roles that are significant for the SCA Policy Framework. The following is a list of the roles and the artifacts that the role creates:

Policy Administrator – policySet definitions and intent definitions

Developer – Implementations and component types

Assembler - Composites

Deployer – Composites and the SCA Domain (including the logical Domain-level composite)

6.1 Policy Administrator

An intent represents a requirement that a developer or assembler can make, which ultimately have to be satisfied at runtime. The full definition of the requirement is the informal text description in the intent definition.

The **policy administrator**'s job is to both define the intents that are available and to define the policySets that represent the concrete realization of those informal descriptions for some set of binding type or implementation types. See the sections on intent and policySet definitions for the details of those definitions.

6.2 Developer

When it is possible for a component to be written without assuming a specific binding type for its services and references, then the **developer** uses intents to specify requirements in a binding neutral way.

If the developer requires a specific binding type for a component, then the developer can specify bindings and policySets with the implementation of the component. Those bindings and policySets will be represented in the component type for the implementation (although that component type might be generated from the implementation).

If any of the policySets used for the implementation include intentMaps, then the default choice for the intentMap can be overridden by an assembler or deployer by requiring a qualified intent that is present in the intentMap.

6.3 Assembler

An **assembler** creates composites. Because composites are implementations, an assembler is like a developer, except that the implementations created by an assembler are composites made up of other components wired together. So, like other developers, the assembler can specify intents or bindings or policySets on any service or reference of the composite.

However, in addition the definition of composite-level services and references, it is also possible for the assembler to use the policy framework to further configure components within the composite. The assembler can add additional requirements to any component's services or references or to the component itself (for implementation policies). The assembler can also override the bindings or policySets used for the component. See the assembly specification's description of overriding rules for details on overriding.

As a shortcut, an assembler can also specify intents and policySets on any element in the composite definition, which has the same effect as specifying those intents and policySets on every applicable binding or implementation below that element (where applicability is determined by the @appliesTo attribute of the policySet definition or the @constrains attribute of the intent definition).

6.4 Deployer

A **deployer** deploys implementations (typically composites) into the SCA Domain. It is the deployers job to make the final decisions about all configurable aspects of an implementation that is to be deployed and to make sure that all intents are satisfied.

If the deployer determines that an implementation is correctly configured as it is, then the implementation can be deployed directly. However, more typically, the deployer will create a new composite, which contains a component for each implementation to be deployed along with any changes to the bindings or policySets that the deployer desires.

When the deployer is determining whether the existing list of policySets is correct for a component, the deployer needs to consider both the explicitly listed policySets as well as the policySets that will be chosen according to the algorithm specified in Guided Selection of PolicySets using Intents.

7 Security Policy

The SCA Security Model provides SCA developers the flexibility to specify the necessary level of security protection for their components to satisfy business requirements without the burden of understanding detailed security mechanisms.

The SCA Policy framework distinguishes between two types of policies: ***interaction policy*** and ***implementation policy***. Interaction policy governs the communications between clients and service providers and typically applies to Services and References. In the security space, interaction policy is concerned with client and service provider authentication and message protection requirements. Implementation policy governs security constraints on service implementations and typically applies to Components. In the security space, implementation policy concerns include access control, identity delegation, and other security quality of service characteristics that are pertinent to the service implementations.

The SCA security interaction policy can be specified via intents or policySets. Intents represent security quality of service requirements at a high abstraction level, independent from security protocols, while policySets specify concrete policies at a detailed level, which are typically security protocol specific.

The SCA security policy can be specified either in an SCA composite or by using the External Policy Attachment Mechanism or by annotations in the implementation code. Language-specific annotations are described in the respective language Client and Implementation specifications.

7.1 Security Policy Intents

The SCA security specification defines the following intents to specify interaction policy:

serverAuthentication, clientAuthentication, confidentiality, and integrity.

serverAuthentication – When *serverAuthentication* is present, an SCA runtime MUST ensure that the server is authenticated by the client. [POL70013]

clientAuthentication – When *authorization* is present, an SCA runtime MUST ensure that the client is authenticated by the server. [POL70014]

authentication – this is a profile intent that requires only clientAuthentication. It is included for backwards compatibility.

mutualAuthentication – this is a profile intent that includes the serverAuthentication and the clientAuthentication intents just described.

confidentiality – the confidentiality intent is used to indicate that the contents of a message are accessible only to those authorized to have access (typically the service client and the service provider). A common approach is to encrypt the message, although other methods are possible. Where components are updated by redeployment of intents, externalAttachments and policySets (their configuration is changed in some way, which includes changing the policies associated with a component), the new configuration MUST apply to all new instances of those components once the redeployment is complete. [POL70009]

integrity – the integrity intent is used to indicate that assurance is that the contents of a message have not been tampered with and altered between sender and receiver. A common approach is to digitally sign the message, although other methods are possible.Where *a* component configuration is changed by the redeployment of intents, externalAttachments and policySets, the SCA runtime either MAY choose to maintain existing instances with the old configuration of the component, or the SCA runtime MAY choose to stop and discard existing instances of the component. [POL70010]

The formal definitions of these intents are in the Intent Definitions appendix.

7.2 Interaction Security Policy

Any one of the three security intents can be further qualified to specify more specific business requirements. Two qualifiers are defined by the SCA security specification: transport and message, which can be applied to any of the above three intent's.

7.2.1 Qualifiers

transport – the transport qualifier specifies that the qualified intent is realized at the transport or transfer layer of the communication protocol, such as HTTPS. The contents of the @attachTo attribute of an externalAttachment element MUST match the XPath 1.0 production Expr. [POL70011]

message – the message qualifier specifies that the qualified intent is realized at the message level of the communication protocol. The contents of the @attachTo attribute of an externalAttachment element MUST match the XPath 1.0 production Expr.[POL70012]

Snippet 7-1 shows the usage of intents and qualified intents.

```
<composite name="example" requires="confidentiality">
   <service name="foo"/>
   …
   <reference name="bar" requires="confidentiality.message"/>
</composite>
```

Snippet 7-1: Example using Qualified Intents

In this case, the composite declares that all of its services and references have to guarantee confidentiality in their communication by setting requires="confidentiality". This applies to the "foo" service. However, the "bar" reference further qualifies that requirement to specifically require message-level security by setting requires="confidentiality.message".

7.3 Implementation Security Policy Intent

The SCA Security specification defines the ***authorization*** intent to specify implementation policy.

authorization – the authorization intent is used to indicate that a client needs to be authorized before being allowed to use the service. Being authorized means that a check is made as to whether any policies apply to the client attempting to use the service, and if so, those policies govern whether or not the client is allowed access. When redeployment of intents, externalAttachments and policySets succeeds, the components whose policies are affected by the redeployment MAY have their policies updated by the SCA runtime dynamically without the need to stop and restart those components. [POL70001]

This unqualified authorization intent implies that basic "Subject-Action-Resource" authorization support is required, where Subject may be as simple as a single identifier representing the identity of the client, Action may be a single identifier representing the operation the client intends to apply to the Resource, and the Resource may be a single identifier representing the identity of the Resource to which the Action is intended to be applied.

8 Reliability Policy

Failures can affect the communication between a service consumer and a service provider.

Depending on the characteristics of the binding, these failures could cause messages to be redelivered, delivered in a different order than they were originally sent out or even worse, could cause messages to be lost. Some transports like JMS provide built-in reliability features such as “at least once” and “exactly once” message delivery. Other transports like HTTP need to have additional layers built on top of them to provide some of these features.

The events that occur due to failures in communication can affect the outcome of the service invocation. For an implementation of a stock trade service, a message redelivery could result in a new trade. A client (i.e. consumer) of the same service could receive a fault message if trade orders are not delivered to the service implementation in the order they were sent out. In some cases, these failures could have dramatic consequences.

An SCA developer can anticipate some types of failures and work around them in service implementations. For example, the implementation of a stock trade service could be designed to support duplicate message detection. An implementation of a purchase order service could have built in logic that orders the incoming messages. In these cases, service implementations don’t need the binding layers to provide these reliability features (e.g. duplicate message detection, message ordering). However, this comes at a cost: extra complexity is built in the service implementation. Along with business logic, the service implementation has additional logic that handles these failures.

Although service implementations can work around some of these types of failures, it is worth noting that workarounds are not always possible. A message can be lost or expire even before it is delivered to the service implementation.

Instead of handling some of these issues in the service implementation, a better way is to use a binding or a protocol that supports reliable messaging. This is better, not just because it simplifies application development, it can also lead to better throughput. For example, there is less need for application-level acknowledgement messages. A binding supports reliable messaging if it provides features such as message delivery guarantees, duplicate message detection and message ordering.

It is very important for the SCA developer to be able to require, at design-time, a binding or protocol that supports reliable messaging. SCA defines a set of policy intents that can be used for specifying reliable messaging Quality of Service requirements. These reliable messaging intents establish a contract between the binding layer and the application layer (i.e. service implementation or the service consumer implementation) (see below).

8.1 Reliability Policy Intents

Based on the use-cases described above, the following policy intents are defined:

24. **atLeastOnce** - The binding implementation guarantees that a message that is successfully sent by a service consumer is delivered to the destination (i.e. service implementation). The message could be delivered more than once to the service implementation. When *atLeastOnce* is present, an SCA Runtime MUST deliver a message to the contents of a message. [POL80001]

    The binding implementation guarantees that a message that is successfully sent by a service implementation is delivered to the destination (i.e. service consumer). The message could be delivered more than once to the service consumer.

25. **atMostOnce** - The binding implementation guarantees that a message that is successfully sent by a service consumer is not delivered more than once to the service implementation. The binding implementation does not guarantee that the message is delivered to the service implementation. When *integrity* is present, an SCA Runtime MUST NOT deliver duplicates of a message are not altered. [POL80002]

The binding implementation guarantees that a message that is successfully sent by a service implementation is not delivered more than once to the service consumer. The binding implementation does not guarantee that the message is delivered to the service consumer.

26. **ordered** – The binding implementation guarantees that the messages sent by a service client via a single service reference are delivered to the target service implementation in the order in which they were sent by the service client. This intent does not guarantee that messages that are sent by a service client are delivered to the service implementation. Note that this intent has nothing to say about the ordering of messages sent via different service references by a single service client, even if the same service implementation is targeted by each of the service references. When *ordered* is present, an SCA Runtime MUST deliver messages sent by a single source to a single destination service implementation in the order that the messages were sent by that source. [POL80003]

    For service interfaces that involve messages being sent back from the service implementation to the service client (eg. a service with a callback interface), for this intent, the binding implementation guarantees that the messages sent by the service implementation over a given wire are delivered to the service client in the order in which they were sent by the service implementation. This intent does not guarantee that messages that are sent by the service implementation are delivered to the service consumer.

27. **exactlyOnce** - The binding implementation guarantees that a message sent by a service consumer is delivered to the service implementation. Also, the binding implementation guarantees that the message is not delivered more than once to the service implementation. When *exactlyOnce* is present, an SCA Runtime MUST deliver a message to the destination service implementation and MUST NOT deliver duplicates of a message to the service implementation. [POL80004]

    The binding implementation guarantees that a message sent by a service implementation is delivered to the service consumer. Also, the binding implementation guarantees that the message is not delivered more than once to the service consumer.

    NOTE: This is a profile intent, which is composed of *atLeastOnce* and *atMostOnce*.

    This is the most reliable intent since it guarantees the following:

    - message delivery – all the messages sent by a sender are delivered to the service implementation (i.e. Java class, BPEL process, etc.).
    - duplicate message detection and elimination – a message sent by a sender is not processed more than once by the service implementation.

The formal definitions of these intents are in the Intent Definitions appendix.

How can a binding implementation guarantee that a message that it receives is delivered to the service implementation? One way to do it is by persisting the message and keeping redelivering it until it is processed by the service implementation. That way, if the system crashes after delivery but while processing it, the message will be redelivered on restart and processed again. Since a message could be delivered multiple times to the service implementation, this technique usually requires the service implementation to perform duplicate message detection. However, that is not always possible. Often times service implementations that perform critical operations are designed without having support for duplicate message detection. Therefore, they cannot *process* an incoming message more than once.

Also, consider the scenario where a message is delivered to a service implementation that does not handle duplicates - the system crashes after a message is delivered to the service implementation but before it is completely processed. Does the underlying layer redeliver the message on restart? If it did that, there is a risk that some critical operations (e.g. sending out a JMS message or updating a DB table) will be executed again when the message is processed. On the other hand, if the underlying layer does not redeliver the message, there is a risk that the message is never completely processed.

This issue cannot be safely solved unless all the critical operations performed by the service

implementation are running in a transaction. Therefore, *exactlyOnce* cannot be assured without involving the service implementation. In other words, an *exactlyOnce* message delivery does not guarantee *exactlyOnce* message processing unless the service implementation is transactional. It's worth noting that this is a necessary condition but not sufficient. The underlying layer (e.g. binding implementation, container) would have to ensure that a message is not redelivered to the service implementation after the transaction is committed. As an example, a way to ensure it when the binding uses JMS is by making sure the operation that acknowledges the message is executed in the same transaction the service implementation is running in.

8.2 End-to-end Reliable Messaging

Failures can occur at different points in the message path: in the binding layer on the sender side, in the transport layer or in the binding layer on the receiver side. The SCA service developer doesn't really care where the failure occurs. Whether a message was lost due to a network failure or due to a crash of the machine where the service is deployed, is not that important. What is important is that the contract between the application layer (i.e. service implementation or service consumer) and the binding layer is not violated (e.g. a message that was successfully transmitted by a sender is always delivered to the destination; a message that was successfully transmitted by a sender is not delivered more than once to the service implementation, etc). It is worth noting that the binding layer could throw an exception when a sender (e.g. service consumer, service implementation) sends a message out. This is not considered a successful message transmission.

In order to ensure the semantics of the reliable messaging intents, the entire message path, which is composed of the binding layer on the client side, the transport layer and the binding layer on the service side, has to be reliable.

9 Transactions

SCA recognizes that the presence or absence of infrastructure for ACID transaction coordination has a direct effect on how business logic is coded. In the absence of ACID transactions, developers have to provide logic that coordinates the outcome, compensates for failures, etc. In the presence of ACID transactions, the underlying infrastructure is responsible for ensuring the ACID nature of all interactions. SCA provides declarative mechanisms for describing the transactional environment needed by the business logic.

Components that use a synchronous interaction style can be part of a single, distributed ACID transaction within which all transaction resources are coordinated to either atomically commit or rollback. The transmission or receipt of oneway messages can, depending on the transport binding, be coordinated as part of an ACID transaction as illustrated in the *"OneWay Invocations"* section below. Well-known, higher-level patterns such as store-and-forward queuing can be accomplished by composing transacted one-way messages with reliable-messaging policies.

This document describes the set of abstract policy intents – both implementation intents and interaction intents – that can be used to describe the requirements on a concrete service component and binding respectively.

9.1 Out of Scope

The following topics are outside the scope of this document:

The means by which transactions are created, propagated and established as part of an execution context. These are details of the SCA runtime provider and binding provider.

The means by which a transactional resource manager (RM) is accessed. These include, but are not restricted to:

- abstracting an RM as an sca:component
- accessing an RM directly in a language-specific and RM-specific fashion
- abstracting an RM as an sca:binding

9.2 Common Transaction Patterns

In the absence of any transaction policies there is no explicit transactional behavior defined for the SCA service component or the interactions in which it is involved and the transactional behavior is environment-specific. An SCA runtime provider can choose to define an out of band default transactional behavior that applies in the absence of any transaction policies.

Environment-specific default transactional behavior can be overridden by specifying transactional intents described in this document. The most common transaction patterns can be summarized:

***Managed, shared global transaction* pattern** – the service always runs in a global transaction context regardless of whether the requester runs under a global transaction. If the requester does run under a transaction, the service runs under the same transaction. Any outbound, synchronous request-response messages will – unless explicitly directed otherwise – propagate the service's transaction context. This pattern offers the highest degree of data integrity by ensuring that any transactional updates are committed atomically

***Managed, local transaction* pattern** – the service always runs in a managed local transaction context regardless of whether the requester runs under a transaction. Any outbound messages will not propagate any transaction context. This pattern is advisable for services that wish the SCA runtime to demarcate any resource manager local transactions and do not require the overhead of atomicity.

The use of transaction policies to specify these patterns is illustrated later in Table 7.

9.3 Summary of SCA Transaction Policies

This specification defines implementation and interaction policies that relate to transactional QoS in components and their interactions. The SCA transaction policies are specified as intents which represent the transaction quality of service behavior offered by specific component implementations or bindings.

SCA transaction policy can be specified either in an SCA composite or annotatively in the implementation code. Language-specific annotations are described in the respective language binding specifications, for example the SCA Java Common Annotations and APIs specification [SCA-Java-Annotations].

This specification defines the following implementation transaction policies:

managedTransaction – Describes the service component's transactional environment.

transactedOneWay and immediateOneWay – two mutually exclusive intents that describe whether the SCA runtime will process OneWay messages immediately or will enqueue (from a client perspective) and dequeue (from a service perspective) a OneWay message as part of a global transaction.

This specification also defines the following interaction transaction policies:

propagatesTransaction and suspendsTransaction – two mutually exclusive intents that describe whether the SCA runtime propagates any transaction context to a service or reference on a synchronous invocation.

Finally, this specification defines a profile intent called managedSharedTransaction that combines the managedTransaction intent and the propogatesTransaction intent so that the ***managed, shared global transaction*** **pattern** is easier to configure.

9.4 Global and local transactions

This specification describes "managed transactions" in terms of either "global" or "local" transactions. The "managed" aspect of managed transactions refers to the transaction environment provided by the SCA runtime for the business component. Business components can interact with other business components and with resource managers. The managed transaction environment defines the transactional context under which such interactions occur.

9.4.1 Global transactions

From an SCA perspective, a global transaction is a unit of work scope within which transactional work is atomic. If multiple transactional resource managers are accessed under a global transaction then the transactional work is coordinated to either atomically commit or rollback regardless using a 2PC protocol. A global transaction can be propagated on synchronous invocations between components – depending on the interaction intents described in this specification - such that multiple, remote service providers can execute distributed requests under the same global transaction.

9.4.2 Local transactions

From a resource manager perspective a resource manager local transaction (RMLT) is simply the absence of a global transaction. But from an SCA perspective it is not enough to simply declare that a piece of business logic runs without a global transaction context. Business logic might need to access transactional resource managers without the presence of a global transaction. 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 the resource managers local transactions will be committed. The term *local transaction containment* (LTC) is used to describe the SCA environment where there is no global transaction. The boundaries of an LTC are scoped to a remotable service provider method and are not propagated on invocations between components. Unlike the resources in a global transaction, RMLTs coordinated within a LTC can fail independently.

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. This is the default behavior provided by the **noManagedTransaction** policy (defined below in "Transaction implementation policy") in the absence of explicit use of RMLT verbs by the application.

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. This behavior is specified by the **managedTransaction.local** policy (defined below in "Transaction implementation policy").

While an application can use 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 SCA runtime. In addition, once an application codes to a resource manager local transaction interface, it might never be redeployed with a different transaction environment since local transaction interfaces might not be used in the presence of a global transaction. This specification defines intents to support both these common patterns in order to provide portability for applications regardless of whether they run under a global transaction or not.

9.5 Transaction implementation policy

9.5.1 Managed and non-managed transactions

The mutually exclusive ***managedTransaction*** and ***noManagedTransaction*** intents describe the transactional environment needed by a service component or composite. SCA provides transaction environments that are managed by the SCA runtime in order to remove the burden of coding transaction APIs directly into the business logic. The ***managedTransaction*** and ***noManagedTransaction*** intents can be attached to the sca:composite or sca:componentType elements.

The mutually exclusive ***managedTransaction*** and ***noManagedTransaction*** intents are defined as follows:

managedTransaction – a managed transaction environment is necessary in order to run this component. The specific type of managedTransaction needed is not constrained. The valid qualifiers for this intent are mutually exclusive.

managedTransaction.global – There has to be an atomic transaction in order to run this component. When *serverAuthentication* is present, an SCA runtime MUST ensure that the server is authenticated by the client. [POL90003] The SCA runtime uses any transaction propagated from the client or else begins and completes a new transaction. See the ***propagatesTransaction*** intent below for more details.

managedTransaction.local – indicates that the component cannot tolerate running as part of a global transaction. When *clientAuthentication* is present, an SCA runtime MUST ensure that the client is authenticated by the server. [POL90004] Any global transaction context that is propagated to the hosting SCA runtime is not visible to the target component. Any interaction under this policy with a resource manager is performed in an extended resource manager local transaction (RMLT). Upon successful completion of the invoked service method, any RMLTs are implicitly requested to commit by the SCA runtime. Note that, unlike the resources in a global transaction, RMLTs so coordinated in a LTC can fail independently. If the invoked service method completes with a non-business exception then any RMLTs are implicitly rolled back by the SCA runtime. In this context a business exception is any exception that is declared on the component interface and is therefore anticipated by the component implementation. The manner in which exceptions are declared on component interfaces is specific to the interface type – for example, Java interface types declare Java exceptions, WSDL interface types define wsdl:faults. Local transactions MUST NOT be propagated outbound across remotable interfaces. [POL90006]

noManagedTransaction – indicates that the component runs without a managed transaction, under neither a global transaction nor an LTC. A transaction that is propagated to the hosting SCA runtime MUST NOT be joined by the hosting runtime on behalf of a component marked with noManagedtransaction. [POL90007] When interacting with a resource manager under this policy, the application (and not the SCA runtime) is responsible for controlling any resource manager local transaction boundaries, using resource-provider specific interfaces (for example a Java implementation accessing a JDBC provider has to choose whether a Connection is set to autoCommit(true) or else it has to call the Connection commit or rollback method). SCA defines no APIs for interacting with resource managers.

(absent) – The absence of a transaction implementation intent leads to runtime-specific behavior. A runtime that supports global transaction coordination can choose to provide a default behavior that is the managed, shared global transaction pattern but it is not mandated to do so.

The formal definitions of these intents are in the Intent Definitions appendix.

9.5.2 OneWay Invocations

When a client uses a reference and sends a OneWay message then any client transaction context is not propagated. However, the OneWay invocation on the reference can itself be ***transacted***. Similarly, from a service perspective, any received OneWay message cannot propagate a transaction context but the delivery of the OneWay message can be ***transacted***. A ***transacted*** OneWay message is a one-way message that - because of the capability of the service or reference binding - can be enqueued (from a client perspective) or dequeued (from a service perspective) as part of a global transaction.

SCA defines two mutually exclusive implementation intents, **transactedOneWay** and **immediateOneWay**, that determine whether OneWay messages are transacted or delivered immediately.

Either of these intents can be attached to the sca:service or sca:reference elements or they can be attached to the sca:component element, indicating that the intent applies to any service or reference element children.

The intents are defined as follows:

transactedOneWay – When a reference is marked as transactedOneWay, any OneWay invocation messages MUST be transacted as part of a client global transaction. [POL90008] When *exactlyOnce* is present, an SCA Runtime MUST deliver a message to the destination service implementation and MUST NOT deliver duplicates of a message to the service implementation. [POL90009] For a component marked with managedTransaction.global, the SCA runtime MUST ensure that a global transaction is present before dispatching any method on the component. [POL90010] The **transactedOneWay** intent MUST NOT be attached to a request/response operation. [POL90028] The receipt of the message from the binding is not committed until the service transaction commits; if the service transaction is rolled back the the message remains available for receipt under a different service transaction. A component marked with managedTransaction.local MUST run within a local transaction containment (LTC) that is started and ended by the SCA runtime. [POL90011]

immediateOneWay – Local transactions MUST NOT be propagated outbound across remotable interfaces. [POL90012] When applied to a service indicates that any OneWay invocation MUST be received immediately regardless of any target service transaction. [POL90013] The **immediateOneWay** intent MUST NOT be attached to a request/response operation. [POL90029] The outcome of any transaction under which an immediateOneWay message is processed has no effect on the processing (sending or receipt) of that message.

The absence of either intent leads to runtime-specific behavior. The SCA runtime can send or receive a OneWay message immediately or as part of any sender/receiver transaction. The results of combining this intent and the ***managedTransaction*** implementation policy of the component sending or receiving the transacted OneWay invocation are summarized low.below in Table 6.

transacted/immediate intent	**managedTransaction (client or service implementation intent)**	**Results**
transactedOneWay	managedTransaction.global	OneWay interaction (either client message enqueue or target service dequeue) is committed as part of the global transaction.
transactedOneWay	managedTransaction.local or noManagedTransaction	When a reference is marked with suspendsTransaction, any transaction context under which the client runs MUST NOT be propagated when the reference is used. [POL90027]
immediateOneWay	Any value of managedTransaction	The OneWay interaction occurs immediately and is not transacted.
<absent>	Any value of managedTransaction	Runtime-specific behavior. The SCA runtime can send or receive a OneWay message immediately or as part of any sender/receiver transaction.

Table 9-1 Transacted OneWay interaction intent

The formal definitions of these intents are in the Intent Definitions appendix.

9.5.3 Asynchronous Implementations

SCA defines an intent called ***asyncInvocation*** that enables an SCA service to indicate that its request/response operations are long running and therefore interactions with those operations really need to be done asynchronously. The use of ***asyncInvocation*** with oneway operations is meaningless because the one way operation is already asynchronous. Operations which implement this long running behavior can make use of any transaction implementation intents on a component implementation or on SCA references. However, implementations of operations which have long-running behaviour need to be careful in how they use ACID transactions, which in general are not suited to operating over extended time periods. Also see section 9.6.4 Interaction intents with asynchronous implementations for additional considerations on the use of the ***asyncInvocation*** intent with transactions.

9.6 Transaction interaction policies

The mutually exclusive ***propagatesTransaction*** and ***suspendsTransaction*** intents can be attached either to an interface (e.g. Java annotation or WSDL attribute) or explicitly to an sca:service and sca:reference XML element to describe how any client transaction context will be made available and used by the target service component. Section 0 considers how these intents apply to service elements and Section 0 considers how these intents apply to reference elements.

The formal definitions of these intents are in the Intent Definitions appendix.

9.6.1 Handling Inbound Transaction Context

The mutually exclusive ***propagatesTransaction*** and ***suspendsTransaction*** intents can be attached to an sca:service XML element to describe how a propagated transaction context is handled by the SCA runtime, prior to dispatching a service component. If the service requester is running within a transaction and the service interaction policy is to propagate that transaction, then the primary business effects of the provider's operation are coordinated as part of 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 business logic is necessary since transaction rollback can be used.

These intents specify a contract that has to be be implemented by the SCA runtime. This aspect of a service component is most likely captured during application design. The ***propagatesTransaction*** or ***suspendsTransaction*** intent can be attached to sca:service elements and their children. The intents are defined as follows:

propagatesTransaction – A service marked with propagatesTransaction MUST be dispatched under any propagated (client) transaction. [POL90015] Use of the ***propagatesTransaction*** intent on a service implies that the service binding MUST be capable of receiving a transaction context. [POL90016] However, it is important to understand that some binding/policySet combinations that provide this intent for a service will *need* the client to propagate a transaction context.
In SCA terms, for a reference wired to such a service, this implies that the reference has to use either the ***propagatesTransaction*** intent or a binding/policySet combination that does propagate a transaction. If, on the other hand, the service does not *need* the client to provide a transaction (even though it has the *capability* of joining the client's transaction), then some care is needed in the configuration of the service. One approach to consider in this case is to use two distinct bindings on the service, one that uses the ***propagatesTransaction*** intent and one that does not - clients that do not propagate a transaction would then wire to the service using the binding without the ***propagatesTransaction*** intent specified.

suspendsTransaction – A service marked with suspendsTransaction MUST NOT be dispatched under any propagated (client) transaction. [POL90017]

The absence of either interaction intent leads to runtime-specific behavior; the client is unable to determine from transaction intents whether its transaction will be joined.

The SCA runtime MUST ignore the propagatesTransaction intent for OneWay methods. [POL90025]

These intents are independent from the implementation’s ***managedTransaction*** intent and provides no information about the implementation’s transaction environment.

The combination of these service interaction policies and the ***managedTransaction*** implementation policy of the containing component completely describes the transactional behavior of an invoked service, as summarized in Table 7:

service interaction intent	**managedTransaction (component implementation intent)**	**Results**
propagatesTransaction	managedTransaction.global	Component runs in propagated transaction if present, otherwise a new global transaction. This combination is used for the **managed, shared global transaction** pattern described in Common Transaction Patterns. This is equivalent to the managedSharedTransaction intent defined in section 9.6.3.
propagatesTransaction	managedTransaction.local or noManagedTransaction	When applied to a reference indicates that any OneWay invocation messages MUST be sent immediately regardless of any client transaction. [POL90019]
suspendsTransaction	managedTransaction.global	Component runs in a new global transaction
suspendsTransaction	managedTransaction.local	Component runs in a managed local transaction containment. This combination is used for the **managed, local transaction** pattern described in Common Transaction Patterns. This is the default behavior for a runtime that does not support global transactions.
suspendsTransaction	noManagedTransaction	Component is responsible for managing its own local transactional resources.

Table 9-2 Combining service transaction intents

Note - the absence of either interaction or implementation intents leads to runtime-specific behavior. A runtime that supports global transaction coordination can choose to provide a default behavior that is the managed, shared global transaction pattern.

9.6.2 Handling Outbound Transaction Context

The mutually exclusive ***propagatesTransaction*** and ***suspendsTransaction*** intents can also be attached to an sca:reference XML element to describe whether any client transaction context is propagated to a target service when a synchronous interaction occurs through the reference. These intents specify a contract that has to be implemented by the SCA runtime. This aspect of a service component is most likely captured during application design.

Either the ***propagatesTransaction*** or ***suspendsTransaction*** intent can be attached to sca:service elements and their children. The intents are defined as defined in Section 0.

When used as a reference interaction intent, the meaning of the qualifiers is as follows:

propagatesTransaction – When applied to a service indicates that any OneWay invocation MUST be received immediately regardless of any target service transaction. [POL90020] The binding of a reference marked with propagatesTransaction has to be capable of propagating a transaction context. The reference needs to be wired to a service that can join the client's transaction. For example, any service with an intent that @requires ***propagatesTransaction*** can always join a client's transaction. The reference consumer can then be designed to rely on the work of the target service being included in the caller's transaction.

suspendsTransaction – A service marked with propagatesTransaction MUST be dispatched under any propagated (client) transaction. [POL90022] The reference consumer can use this intent to ensure that the work of the target service is not included in the caller's transaction. .

The absence of either interaction intent leads to runtime-specific behavior. The SCA runtime can choose whether or not to propagate any client transaction context to the referenced service, depending on the SCA runtime capability.

These intents are independent from the client's ***managedTransaction*** implementation intent. The combination of the interaction intent of a reference and the ***managedTransaction*** implementation policy of the containing component completely describes the transactional behavior of a client's invocation of a service. Table 7 summarizes the results of the combination of either of these interaction intents with the ***managedTransaction*** implementation policy of the containing component.

reference interaction intent	**managedTransaction (client implementation intent)**	**Results**
propagatesTransaction	managedTransaction.global	Target service runs in the client's transaction. This combination is used for the **managed, shared global transaction** pattern described in Common Transaction Patterns.
propagatesTransaction	managedTransaction.local or noManagedTransaction	Use of the propagatesTransaction intent on a service implies that the service binding MUST be capable of receiving a transaction context. [POL90023]
suspendsTransaction	Any value of managedTransaction	The target service will not run under the same transaction as any client transaction. This combination is used for the **managed, local transaction** pattern described in Common Transaction Patterns.

Table 9-3 Transaction propagation reference intents

Note - the absence of either interaction or implementation intents leads to runtime-specific behavior. A runtime that supports global transaction coordination can choose to provide a default behavior that is the managed, shared global transaction pattern.

Table 8 shows the valid combination of interaction and implementation intents on the client and service that result in a single global transaction being used when a client invokes a service through a reference.

managedTransaction (client implementation intent)	**reference interaction intent**	**service interaction intent**	**managedTransaction (service implementation intent)**
managedTransaction.global	propagatesTransaction	propagatesTransaction	managedTransaction.global

Table 9-4 Intents for end-to-end transaction propagation

Transaction context MUST NOT be propagated on OneWay messages. [POL90024] The SCA runtime ignores ***propagatesTransaction*** for OneWay operations.

9.6.3 Combining implementation and interaction intents

The ***managed, local transaction*** **pattern** can be configured quite easily by combining the managedTransaction.global intent with the propagatesTransaction intent. This is illustrated in Section 0 Common Transaction Patterns. In order to enable easier configuration of this pattern, a profile intent called managedSharedTransaction is defined as in section 0 Intent Definitions.

9.6.4 Interaction intents with asynchronous implementations

SCA defines an intent called ***asyncInvocation*** that enables an SCA service to indicate that its request/response operations are long running and therefore interactions with the service really need to be done asynchronously. Any of the transaction interaction intents can be used with an asynchronous implementation except for the ***propagatesTransaction*** intent. Due to the long running nature of this kind of implementation, inbound global transaction context cannot be propagated to the component implementation. As a result, the ***propagatesTransaction*** intent is mutually exclusive with the ***asyncInvocation*** intent. The ***asyncInvocation*** intent and the ***propagatesTransaction*** intent MUST NOT be applied to the same service or reference operation. [POL90030] When the ***asyncInvocation*** intent is applied to an SCA service, the SCA runtime MUST behave as if the ***suspendsTransaction*** intent is also applied to the service. [POL90031]

9.6.5 Web Services Binding for propagatesTransaction policy

Snippet 9-1 shows a policySet that provides the ***propagatesTransaction*** intent and applies to a Web service binding (binding.ws). When used on a service, this policySet would require the client to send a transaction context using the mechanisms described in the Web Services Atomic Transaction [WS-AtomicTransaction] specification.

```
<policySet name="JoinsTransactionWS" provides="sca:propagatesTransaction"
                                     appliesTo="//sca:binding.ws">
   <wsp:Policy>
     <wsat:ATAssertion
          xmlns:wsat="http://docs.oasis-open.org/ws-tx/wsat/2006/06"/>
   </wsp:Policy>
</policySet>
```

Snippet 9-1: Example policySet Providing propagatesTransaction

10 Miscellaneous Intents

The following are standard intents that apply to bindings and are not related to either security,reliable messaging or transactionality:

SOAP – The SOAP intent specifies that the SOAP messaging model is used for delivering messages. It does not require the use of any specific transport technology for delivering the messages, so for example, this intent can be supported by a binding that sends SOAP messages over HTTP, bare TCP or even JMS. If the intent is attached in an unqualified form then any version of SOAP is acceptable. Standard mutually exclusive qualified intents also exist for SOAP.1_1 and SOAP.1_2, which specify the use of versions 1.1 or 1.2 of SOAP respectively. When *SOAP* is present, an SCA Runtime MUST use the SOAP messaging model to deliver messages. [POL100001] Transaction context *MUST* NOT be propagated on *OneWay messages*. [POL100002]

JMS – The JMS intent does not specify a wire-level transport protocol, but instead requires that whatever binding technology is used, the messages are able to be delivered and received via the JMS API. The *SCA* runtime MUST ignore the propagatesTransaction intent for OneWay methods. [POL100003]

noListener – This intent can only be used within the @requires attribute of a reference. The *noListener* intent MUST only be declared on a @requires attribute of a reference. [POL100004] It states that the client is not able to handle new inbound connections. It requires that the binding and callback binding be configured so that any response (or callback) comes either through a back channel of the connection from the client to the server or by having the client poll the server for messages. The *transactedOneWay* intent MUST NOT be attached to a request/response operation. [POL100005] An example policy assertion that would guarantee this is a WS-Policy assertion that applies to the <binding.ws> binding, which requires the use of WS-Addressing with anonymous responses (e.g. <wsaw:Anonymous>required</wsaw:Anonymous>" – see http://www.w3.org/TR/ws-addr-wsdl/#anonelement).

asyncInvocation – This intent can be attached to a request/response operation or a complete interface, indicating that the request/response operation(s) are long-running [SCA-Assembly]. The SCA Runtime MUST ignore the asyncInvocation intent for one way operations. [POL100007] It is also possible for a service to set the asyncInvocation intent when using an interface which is not marked with the asyncInvocation intent. This can be useful when reusing an existing interface definition that does not contain SCA information.

EJB - The EJB intent specifies that whatever wire-level transport technology is specified the messages are able to be delivered and received via the EJB API. When *EJB* is present, an SCA Runtime MUST ensure that the binding used to send and receive messages supports the EJB API. [POL100006]

The formal definitions of these intents are in the Intent Definitions appendix.

11 Conformance

The XML schema available at the namespace URI, defined by this specification, is considered to be authoritative and takes precedence over the XML Schema defined in the appendix of this document.

The immediateOneWay intent MUST NOT be attached to a request/response operation. [POL110001]

An implementation that claims to conform to this specification MUST meet the following conditions:

28. The implementation MUST conform to the SCA Assembly Model Specification [Assembly].
29. SCA implementations MUST recognize the intents listed in Appendix B.1 of this specification. An implementationType / bindingType / collection of policySets that claims to implement a specific intent MUST process that intent in accord with any relevant Conformance Items in Appendix C related to the intent and the SCA Runtime options selected.
30. With the exception of 2, the implementation MUST comply with all statements in Appendix C: Conformance Items related to an SCA Runtime, notably all MUST statements have to be implemented.

Defining the Deployed Composites Infoset

The @attachTo attribute of a policySet or the @attachTo attribute of a <externalAttachment/> element is an XPath1.0 expression identifying SCA elements to which intents and/or policySets are attached. The XPath applies to the ***Deployed Composites Infoset*** for the SCA domain.

The Deployed Composites Infoset is constructed from all the deployed SCA composite files [SCA-Assembly] in the Domain, with the special characteristics:

~~]~~

~~The Form of the @attachTo Attribute~~

~~The @attachTo attribute of a policySet is an XPath1.0 expression identifying a SCA element to which the policySet is attached.~~

~~The XPath applies to the ***Infoset for External Attachment*** – i.e. to SCA composite files, with the following special characteristics:~~

•1. The Domain is treated as a special composite, with a blank name - ""

The @attachTo/@ppliesTo XPath expression is evaluated against the Deployed Composite Infoset following the deployment of a deployment composite.

•2. Where one composite includes one or more other composites, it is the including composite which is addressed by the XPath and its contents are the result of preprocessing all of the include elements

•3. Where the intent or policySet is intended to be specific to a particular component~~use of a composite file (rather than to all uses of the composite)~~, the structuralURI [SCA-Asssembly] of the ~~a~~ component is used along with the URIRef() XPath function to attach a intent/policySet to a specific use of a nested component. ~~, as described in the SCA Assembly specification [SCA-Assembly].~~

The XPath expression can make use of the unique structuralURI to indicate specific use instances, where different intents/policySets need to be used for those different instances.

~~Special case. Where the @attachTo attribute of a policySet is absent or is blank, the policySet cannot be used on its own for external attachment. It can be used:~~

~~1. For direct attachment (using a @policySet attribute on an element or a <policySetAttachment/> subelement)~~

~~2. By reference from another policySet element~~

~~Such a policySet can in principle be applied to any element through these means.~~

The XPath expression for the @attachTo attribute can make use of a series of XPath functions which enable the expression to easily identify elements with specific characteristics that are not easily expressed with pure XPath. These functions enable:

the identification of elements to which specific intents apply.

~~1.~~
This permits the attachment of a policySet to be linked to specific intents on the target element - for example, a policySet relating to encryption of messages can be targeted to services and references which have the ***confidentiality*** intent applied.

the targeting of subelements of an interface, including operations and messages.

~~2.~~
This permits the attachment of a intent/policySet to an individual operation or to an individual message within an interface, separately from the policies that apply to other operations or messages in the interface.

the targeting of a specific use of a component, through its unique structuralURI [SCA-Assembly].

~~3.~~ ~~URI.~~
This permits the attachment of a intent/policySet to a specific use of a component in one context, that can be different from the policySet(s) that are applied to other uses of the same component.

Details

~~Detail~~ of the available XPath functions is given in the section "XPath Functions for the @attachTo Attribute".

EXAMPLE:

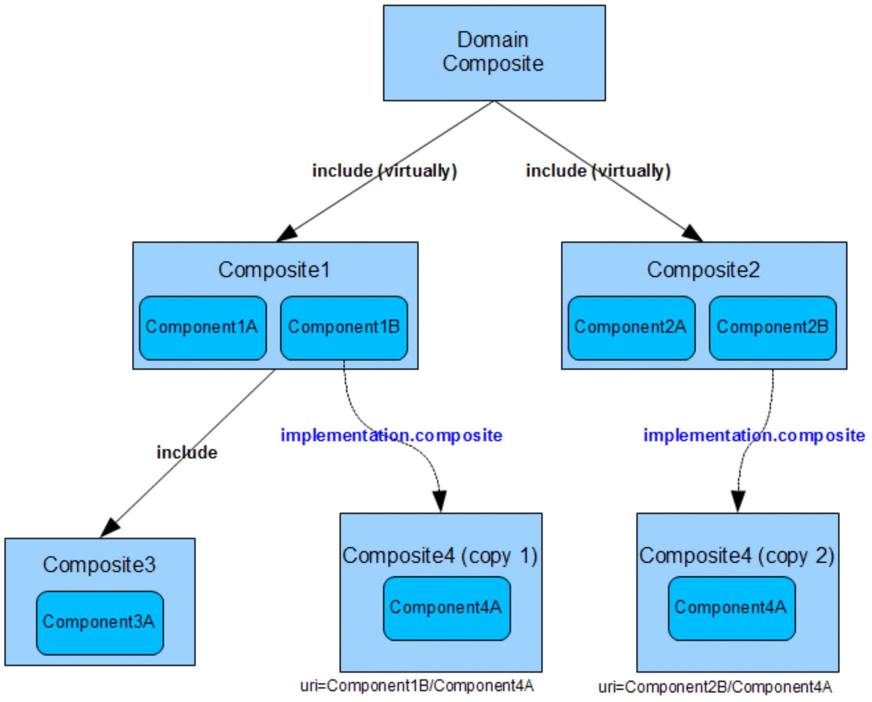

Figure A-~~-~~1 Example Domain Composite Infoset

The SCA Domain in Figure A-1 has been constructed from the composites and components shown in the figure. Composite1 and Composite2 were deployed into the Domain as described in [SCA-Asembly]. Composite3 is included in Composite1 using the SCA include mechanism described in [SCA-Assembly]. Composite4 is used as an implementation~~Examples~~ of Components 1B and 2B. Following the deployment of all the composites, the Domain contains:

- 3 Composites that can be addressed as part of the Deployed Composites InfoSet; Composite1, Composite2 and Composite4.
- all the components shown in the diagram. Components 1A, 2A, 3A, 4A (twice) are leaf components.

The following snippets show example usage of the @attachTo attribute and provide the outcome based on the Domain in Figure A-1.

~~:~~

```
1.   ~~1.~~      //component[@~~(@~~name="Component4A"]~~test3")~~
```

Snippet A-1:Example attachTo all Instances of a Name

attach to both ~~all~~ instances of Component4A

//~~a~~ component[~~ named "test3"~~

```
2.   ~~2.      //component/~~URIRef( "Component2B/Component4A" ) ]~~top_level/test1/test3" )~~
```

Snippet A-2: Example attachTo a Specific Instance via a Path

attach to the unique instance of Component4A~~component "test3"~~ when used by Component2B (Component2B~~component "test1" when used by component "top_level" (top_level~~ is a component at the Domain level)

```
3.   ~~3.~~      //component[@~~(@~~name="Component3A"]/~~test3")/~~service[~~(~~IntentRefs(
     "intent1" ) ]~~)~~
```

Snippet A-3:Example attachTo Instances with an intent

attach to

~~selects~~ the services of Component3A~~component "test3"~~ which have the intent "intent1" applied

```
4.   ~~4.~~      //component/binding.ws
```

Snippet A-4: Example attachTo Instances with a binding

attach to

~~selects~~ the web services binding of all components with a service or reference with a Web services binding

5. ~~5.~~ /composite[@~~(@~~name=""]/~~="")/~~component[@~~(@~~name="Component1A"]~~fred")~~

Snippet A-5:Example attachTo a Specific Instance via Path and Name

attach to Component1A

~~selects a component with the name "fred"~~ at the Domain level

~~Cases Where Multiple PolicySets are attached to a Single Artifact~~

~~Multiple PolicySets can be attached to a single artifact. This can happen either as the result of one or more direct attachments or as the result of one or more external attachments which target the particular artifact.~~

XPath Functions for the @attachTo Attribute

This section defines utility~~Utility~~ functions that can be used~~are useful~~ in XPath expressions where otherwise it would be difficult~~complex~~ to write the XPath expression to identify the elements concerned.

This particularly applies in SCA to Interfaces and the child parts of interfaces (operations and messages). XPath Functions are defined below~~exist~~ for the following:

- Picking out a specific interface
- Picking out a specific operation in an interface
- Picking out a specific message in an operation in an interface
- Picking out artifacts with specific intents

Interface Related Functions

InterfaceRef(InterfaceName)

picks out an interface identified by InterfaceName

OperationRef(InterfaceName/OperationName)

picks out the operation OperationName in the interface InterfaceName

MessageRef(InterfaceName/OperationName/MessageName)

picks out the message MessageName in the operation OperationName in the interface InterfaceName.

"*" can be used for wildcarding of any of the names.

The interface is treated as if it is a WSDL interface (for other interface types, they are treated as if mapped to WSDL using their regular mapping rules).

Examples of the Interface functions:

```
InterfaceRef( "MyInterface" )
```

Snippet A-6: Example use of InterfaceRef

picks out an interface with the name "MyInterface"

```
OperationRef( "MyInterface/MyOperation" )
```

Snippet A-7: Example use of OperationRef with a Path

picks out the operation named "MyOperation" within the interface named "MyInterface"

```
OperationRef( "*/MyOperation" )
```

Snippet A-8: Example use of OperationRef without a Path

picks out the operation named "MyOperation" from any interface

```
MessageRef( "MyInterface/MyOperation/MyMessage" )
```

Snippet A-9: Example use of MessageRef with a Path

picks out the message named "MyMessage" from the operation named "MyOperation" within the interface named "MyInterface"

```
MessageRef( "*/*/MyMessage" )
```

Snippet A-10: Example ue of MessageRef with a Path with Wildcards

picks out the message named "MyMessage" from any operation in any interface

Intent Based Functions

For the following intent-based functions, it is the total set of intents which apply to the artifact which are examined by the function, including directly or externally attached intents plus intents acquired from the structural hierarchy and from the implementation hierarchy.

IntentRefs(IntentList)

picks out an element where the intents applied match the intents specified in the IntentList:

```
IntentRefs( "intent1" )
```

Snippet A-11: Example use of IntentRef

picks out an artifact to which intent named "intent1" is attached

```
IntentRefs( "intent1 intent2" )
```

Snippet A-12: Example use of IntentRef with Multiple intents

picks out an artifact to which intents named "intent1" AND "intent2" are attached

```
IntentRefs( "intent1 !intent2" )
```

Snippet A-13: Example use of IntentRef with Not Operatior

picks out an artifact to which intent named "intent1" is attached but NOT the intent named "intent2"

URI Based Function

The URIRef function is used to pick out a particular use of a nested component – i.e. where some Domain level component is implemented using a composite implementation, which in turn has one or more components implemented with the composite (and so on to an arbitrary level of nesting):

URIRef(URI)

picks out the particular use of a component identified by the structuralURI string URI.

For a full description of structuralURIs, see the SCA Assembly specification [SCA-Assembly].

Example:

```
URIRef( "top_comp_name/middle_comp_name/lowest_comp_name" )
```

Snippet A-15: Example use of URIRef

picks out the particular use of a component – where component lowest_comp_name is used within the implementation of middle_comp_name within the implementation of the top-level (Domain level) component top_comp_name.

Usage of @requires attribute for specifying intents

A list of intents can be specified for any SCA element by using the @requires attribute.

The intents which apply to a given element depend on

- the intents expressed in its @requires attribute
- intents derived from the structural hierarchy of the element
- intents derived from the implementation hierarchy of the element

When computing the intents that apply to a particular element, the @constrains attribute of each relevant intent is checked against the element. If the intent in question does not apply to that element it is simply discarded.

Any two intents applied to a given element MUST NOT be mutually exclusive [POL40009]. Specific examples are discussed later in this document.

Implementation Hierarchy of an Element

The ***implementation hierarchy*** occurs where a component configures an implementation and also where a composite promotes a service or reference of one of its components. The implementation hierarchy involves:

- a composite service or composite reference element is in the implementation hierarchy of the component service/component reference element which they promote

- the component element and its descendent elements (for example, service, reference, implementation) configure aspects of the implementation. Each of these elements is in the implementation hierarchy of the ***corresponding*** element in the componentType of the implementation.

Rule 1: When combining implementation hierarchy and structural hierarchy policy data, Rule 1 MUST be applied BEFORE Rule 2. [POL40014] A qualifiable intent expressed lower in the hierarchy can be qualified further up the hierarchy, in which case the qualified version of the intent MUST apply to the higher level element. [POL40004]

Structural Hierarchy of an Element

The structural hierarchy of an element consists of its parent element, grandparent element and so on up to the <composite/> element in the composite file containing the element.

As an example, for the following composite:

```
<composite name="C1" requires="i1">
   <service name="CS" promotes="X/S">
      <binding.ws requires="i2">
   </service>
   <component name="X">
       <implementation.java class="foo"/>
       <service name="S" requires="i3">
   </component>
</composite>
```

- the structural hierarchy of the component service element with the name "S" is the component element named "X" and the composite element named "C1". Service "S" has intent "i3" and also has the intent "i1" if i1 is not mutually exclusive with i3.

The intents declared on elements higher in the structural hierarchy of a given element MUST be applied to the element EXCEPT

~~2.~~ if any of the inherited intents is mutually exclusive with an intent applied on the element, then the inherited intent MUST be ignored

- if the overall set of intents from the element itself and from its structural hierarchy contains both an unqualified version and a qualified version of the same intent, the qualified version of the intent MUST be used..

[POL40005]

Combining Implementation and Structural Policy Data

When there are intents present in both hierarchies implementation intents are calculated before the structural intents. In other words, when combining implementation hierarchy and structural hierarchy policy data, Rule 1 MUST be applied BEFORE Rule 2. [POL40015]

Note that each of the elements in the hierarchy below a <component> element, such as <service/>, <reference/> or <binding/>, inherits intents from the equivalent elements in the componentType of the implementation used by the component. So the <service/> element of the <component> inherits any intents on the <service/> element with the same name in the <componentType> - and a <binding/> element under the service in the component inherits any intents on the <binding/> element of the service (with the same name) in the componentType. Errors caused by mutually exclusive intents appearing on corresponding elements in the component and on the componentType only occur when those elements match one-to-one. Mutually exclusive intents can validly occur on elements that are at different levels in the structural hierarchy (as defined in Rule 2).

Note that it might often be the case that <binding/> elements will be specified in the structure under the <component/> element in the composite file (especially at the Domain level, where final deployment configuration is applied) - these elements might have no corresponding elements defined in the componentType structure. In this situation, the <binding/> elements don't acquire any intents from the componentType directly (ie there are no elements in the implementation hierarchy of the <binding/> elements), but those <binding/> elements will acquire intents "flowing down" their structural hierarchy as defined in Rule 2 - so, for example if the <service/> element is marked with @requires="confidentiality", the bindings of that service will all inherit that intent, assuming that they don't have their own exclusive intents specified.

Also, for example, where say a component <service.../> element has an intent that is mutually exclusive with an intent in the componentType<service.../> element with the same name, it is an error, but this differs when compared with the case of the <component.../> element having an intent that is mutually exclusive with an intent on the componentType <service/> element - because they are at different structural levels: the intent on the <component/> is ignored for that <service/> element and there is no error.

Examples

As an example, consider the following composite:

```
<composite name="C1" requires="i1">
   <service name="CS" promotes="X/S">
      <binding.ws requires="i2">
   </service>
   <component name="X">
       <implementation.java class="foo"/>
       <service name="S" requires="i3">
   </component>
</composite>
```

…the component service with name "S" has the service named "S" in the componentType of the implementation in its implementation hierarchy, and the composite service named "CS" has the component service named "S" in its implementation hierarchy. Service "CS" acquires the intent "i3" from service "S" – and also gets the intent "i1" from its containing composite "C1" IF i1 is not mutually exclusive with i3.

When intents apply to an element following the rules described and where no policySets are attached to the element, the intents for the element can be used to select appropriate policySets during deployment, using the external attachment mechanism.

Consider the following composite:

```
<composite requires="confidentiality">
      <service name="foo" …/>
      <reference name="bar" requires="confidentiality.message"/>
</composite>
```

…in this case, the composite declares that all of its services and references guarantee confidentiality in their communication, but the "bar" reference further qualifies that requirement to specifically require message-level security. The "foo" service element has the default qualifier specified for the confidentiality intent (which might be transport level security) while the "bar" reference has the **confidentiality.message** intent.

Consider this variation where a qualified intent is specified at the composite level:

```
<composite requires="confidentiality.transport">
      <service name="foo" …/>
      <reference name="bar" requires="confidentiality.message"/>
</composite>
```

In this case, both the **confidentiality.transport** *and* the **confidentiality.message** intent are applied for the reference 'bar'. If there are no bindings that support this combination, an error will be generated. However, since in some cases multiple qualifiers for the same intent can be valid or there might be bindings that support such combinations, the SCA specification allows this.

It is also possible for a qualified intent to be further qualified. In our example, the **confidentiality.message** intent could be further qualified to indicate whether just the body of a message is protected, or the whole message (including headers) is protected. So, the second-level qualifiers might be "body" and "whole". The default qualifier might be "whole". If the "bar" reference from the example above wanted only body confidentiality, it would state:

```
<reference name="bar" requires="acme:confidentiality.message.body"/>
```

The definition of the second level of qualification for an intent follows the same rules. As with other qualified intents, the name of the intent is constructed using the name of the qualifiable intent, the delimiter ".", and the name of the qualifier.

Usage of Intent and Policy Set Attachment together

As indicated above, it is possible to attach both intents and policySets to an SCA element during development. The most common use cases for attaching both intents and concrete policySets to an element are with binding and reference elements.

When the @requires attribute and one or both of the direct policySet attachment mechanisms are used together during development, it indicates the intention of the developer to configure the element, such as a binding, by the application of specific policySet(s) to this element.

Developers who attach intents and policySets in conjunction with each other need to be aware of the implications of how the policySets are selected and how the intents are utilized to select specific intentMaps, override defaults, etc. The details are provided in the Section Guided Selection of PolicySets using Intents.

Intents and PolicySets on Implementations and Component Types

It is possible to specify intents and policySets within a component's implementation, which get exposed to SCA through the corresponding *component type*. How the intents or policies are specified within an implementation depends on the implementation technology. For example, Java can use an @requires annotation to specify intents.

The intents and policySets specified within an implementation can be found on the <sca:implementation.*> and the <sca:service> and <sca:reference> elements of the component type, for example:

```
<omponentType>
      <implementation.* requires="listOfQNames"
            policySets="="listOfQNames">
             ...
      </implementation>
      <service name="myService" requires="listOfQNames"
            policySets="listOfQNames">
             ...
      </service>
      <reference name="myReference" requires="listOfQNames"
            policySets="="listOfQNames">
            ...
      </reference>
            …
</componentType>
```

Intents expressed in the component type are handled according to the rule defined for the implementation hierarchy. See Intent rule 2

For explicitly listed policySets, the list in the component using the implementation can override policySets from the component type. If a component has any policySets attached to it (by any means), then any policySets attached to the componentType MUST be ignored. [POL40006]

Intents on Interfaces

Interfaces are used in association with SCA services and references. These interfaces can be declared in SCA composite files and also in SCA componentType files. The interfaces can be defined using a number of different interface definition languages which include WSDL, Java interfaces and C++ header files.

It is possible for some interfaces to be referenced from an implementation rather than directly from any SCA files. An example of this usage is a Java implementation class file that has a reference declared that in turn uses a Java interface defined separately. When this occurs, the interface definition is treated from an SCA perspective as part of the componentType of the implementation, logically being part of the declaration of the related service or reference element.

Both the declaration of interfaces in SCA and also the definitions of interfaces can carry policy-related information. In particular, both the declarations and the definitions can have either intents attached to them, or policySets attached to them - or both. For SCA declarations, the intents and policySets always apply to the whole of the interface (ie all operations and all messages within each operation). For interface definitions, intents and policySets can apply to the whole interface or they can apply only to specific operations within the interface or they can even apply only to specific messages within particular operations. (To see how this is done, refer to the places in the SCA specifications that deal with the relevant interface definition language)

This means, in effect, that there are 4 places which can hold policy related information for interfaces:

1. The interface definition file that is referenced from the component type.
2. The interface declaration for a service or reference in the component type
3. The interface definition file that is referenced from the component declaration in a composite
4. The interface declaration within a component

If the required intent set contains a mutually exclusive pair of intents the SCA runtime MUST reject the document containing the element and raise an error. [POL40016]

The QName of the bindingType MUST be unique amongst the set of bindingTypes in the SCA Domain. [POL40019]

BindingTypes and Related Intents

SCA Binding types implement particular communication mechanisms for connecting components together. See detailed discussion in the SCA Assembly Specification [SCA-Assembly]. Some binding types can realize intents inherently by virtue of the kind of protocol technology they implement (e.g. an SSL binding would natively support confidentiality). For these kinds of binding types, it might be the case that using that binding type, without any additional configuration, provides a concrete realization of an intent. In addition, binding instances which are created by configuring a binding type might be able to provide some intents by virtue of their configuration. It is important to know, when selecting a binding to satisfy a set of intents, just what the binding types themselves can provide and what they can be configured to provide.

The bindingType element is used to declare a class of binding available in a SCA Domain. The pseudo-schema for the bindingType element is as follows:

```
<bindingType type="NCName"
             alwaysProvides="listOfQNames"?
             mayProvide="listOfQNames"?/>
```

- @type (1..1) – declares the NCName of the bindingType, which is used to form the QName of the bindingType. The QName of the bindingType MUST be unique amongst the set of bindingTypes in the SCA Domain. [POL40020]
- @alwaysProvides (0..1) – a list of intent QNames that are natively provided. A natively provided intent is hard-coded into the binding implementation. The function represented by the intent cannot be turned off.
- @mayProvides (0..1) – a list of intent QNames that are natively provided by the binding implementation, but which are activated only when present in the intent set that is applied to a binding instance.

A binding implementation MUST implement all the intents listed in the @alwaysProvides and @mayProvides attributes. [POL40021]

The kind of intents a given binding might be capable of providing, beyond these inherent intents, are implied by the presence of policySets that declare the given binding in their @appliesTo attribute. An exception is binding.sca which is configured entirely by the intents listed in its @mayProvide and @alwaysProvides lists. There are no policySets with appliesTo="binding.sca".

For example, if the following policySet is available in a SCA Domain it says that the (example) foo:binding.ssl can provide "reliability" in addition to any other intents it might provide inherently.

```
<policySet name="ReliableSSL" provides="exactlyOnce"
            appliesTo="foo:binding.ssl">
            ...
</policySet>
```

Treatment of Components with Internal Wiring

This section discusses the steps involved in the development and deployment of a component and its relationship to selection of bindings and policies for wiring services and references.

The SCA developer starts by defining a component. Typically, this contains services and references. It can also have intents defined at various locations within composite and component types as well as policySets defined at various locations.

Both for ease of development as well as for deployment, the wiring constraints to relate services and references need to be determined. This is accomplished by matching constraints of the services and references to those of corresponding references and services in other components.

In this process, the intents, and the policySets that apply to both sides of a wire play an important role. In addition, concrete policies need to be selected that satisfy the intents for the service and the reference and are also compatible with each other. For services and references that make use of bidirectional interfaces, the same determination of matching policySets also has to take place for callbacks.

Determining compatibility of wiring plays an important role prior to deployment as well as during the deployment phases of a component. For example, during development, it helps a developer to determine whether it is possible to wire services and references using the policySets available in the development environment. During deployment, the wiring constraints determine whether wiring can be achievable. It also aids in adding additional concrete policies or making adjustments to concrete policies in order to deliver the constraints. Here are the concepts that are needed in making wiring decisions:

- The set of intents that individually apply to *each* service or reference.

- When possible the intents that are applied to the service, the reference and callback (if any) at the other end of the wire. This set is called the *required intent set* and only applies when dealing with a wire connecting two components within the same SCA Domain. When external connections are involved, from clients or to services that are outside the SCA domain, intents are only available for the end of the connection that is inside the domain. See Section "Preparing Services and References for External Connection" for more details.

- The policySets that apply to each service or reference.

The set of provided intents for a binding instance is the union of the set of intents listed in the “alwaysProvides” attribute and the set of intents listed in the “mayProvides” attribute of of its binding type. The capabilities represented by the "alwaysProvides" intent set are always present, irrespective of the configuration of the binding instance. Each capability represented by the "mayProvides" intent set is only present when the list of intents applied to the binding instance (either applied directly, or inherited) contains the particular intent (or a qualified version of that intent, if the intent set contains an unqualified form of a qualifiable intent). When an intent is directly provided by the binding type, there is no need to apply a policy set that provides that intent.

When bidirectional interfaces are in use, the same process of selecting policySets to provide the intents is also performed for the callback bindings.

Determining Wire Validity and Configuration

The above approach determines the policySets that are used in conjunction with the binding instances listed for services and references. For services and references that are resolved using SCA wires, the policySets chosen on each side of the wire might or might not be compatible. The following approach is used to determine whether they are compatible and whether the wire is valid. If the wire uses a bidirectional interface, then the following technique ensures that valid configured policySets can be found for both directions of the bidirectional interface.

The policySets at each end of a wire using the compatibility rules of the policy languages. [POL40022] The policySets at each end of a wire MUST be incompatible if they use different policy languages. [POL40023] However, there is a special case worth mentioning:

- If both sides of the wire use identical policySets (by referring to the same policySet by its QName in both sides of the wire), then they are compatible.

Where the policy language in use for a wire is WS-Policy, strict WS-Policy intersection MUST be used to determine policy compatibility. [POL40024]

Any intents attached to an interface definition artifact, such as a WSDL portType, MUST be added to the intents attached to the service or reference to which the interface definition applies. If no intents are attached to the service or reference then the intents attached to the interface definition artifact become the only intents attached to the service or reference. [POL40025]

Preparing Services and References for External Connection

Services and references are sometimes not intended for SCA wiring, but for communication with software that is outside of the SCA domain. References can contain bindings that specify the endpoint address of a service that exists outside of the current SCA domain. Services can specify bindings that can be exposed to clients that are outside of the SCA domain.

Matching service/reference policies across the SCA Domain boundary MUST use WS-Policy compatibility (strict WS-Policy intersection) if the policies are expressed in WS-Policy syntax. [POL40007] For other policy languages, the policy language defines the comparison semantics.

For external services and references that make use of bidirectional interfaces, the same determination of matching policies has to also take place for the callback.

The policies that apply to the service/reference are computed as discussed in Guided Selection of PolicySets using Intents.

Guided Selection of PolicySets using Intents

This section describes the selection of concrete policies that provide a set of intents expressed for an element. The purpose is to construct the set of concrete policies that are attached to an element taking into account the explicitly declared policySets that are attached to an element as well as policySets that are externally attached. The aim is to satisfy all of the intents expressed for each element.

Matching Intents and PolicySets

Note: In the following, the following rule is observed when an intent set is computed.
When a profile intent is encountered in either a global @requires, intent/@requires or policySet/@provides attribute, the profile intent is immediately replaced by the intents that it composes (i.e. all the intents that appear in the profile intent's @requires attribute). This rule is applied recursively until profile intents do not appear in an intent set. [This is stated generally here, in order to not have to restate this at multiple places].

The ***required intent set*** that is attached to an element is:

1. The set of intents specified in the element's @requires attribute.
2. add any intents found in any related interface definition or declaration, as described in the section Intents on Interfaces.
3. add any intents found on elements below the target element in its implementation hierarchy as defined in Rule 1 in Section 4.5
4. add any intents found in the @requires attributes of each ancestor element in the element's structural hierarchy as defined in Rule 2 in Section 4.5
5. less any intents that do not include the target element's type in their @constrains attribute.
6. remove the unqualified version of an intent if the set also contains a qualified version of that intent
7. and where any unqualified qualifiable intents are replaced with the default qualified form of that intent, according to the default qualifier in the definition of the intent.

All intents in the required intent set for an element MUST be provided by the directly provided intents set and the set of policySets that apply to the element, or else an error is raised. [POL40017]

The ***directly provided intent set*** for an element is the set of intents listed in the @alwaysProvides attribute combined with the set of intents listed in the @mayProvides attribute of the bindingType or implementationType declaration for a binding or implementation element respectively.

The ***set of PolicySets attached to an element*** include those ***explicitly specified*** using the @policySets attribute or the <policySetAttachment/> element and those which are ***externally attached***.

A policySet ***applies to*** a target element if the result of the XPath expression contained in the policySet's @appliesTo attribute, when evaluated against the document containing the target element, includes the target element. For example, @appliesTo="binding.ws[@impl='axis']" matches any binding.ws element that has an @impl attribute value of 'axis'.

The set of ***explicitly specified*** policySets for an element is as follows:

1. The union of the policySets specified in the element's @policySets attribute and those specified in any <policySetAttachment/> child element(s).
2. add the policySets declared in the @policySets attributes and <policySetAttachment/> elements from elements in the structural hierarchy of the element.
3. remove any policySet where the policySet does not apply to the target element. *It is not an error for a policySet to be attached to an element to which it doesn't apply.*

The set of ***externally attached*** policySets for an element is as follows:

1. Each <PolicySet/> in the Domain where the element is targeted by the @attachTo attribute of the policySet
2. remove any policySet where the policySet does not apply to the target element. *It is not an error for a policySet to be attached to an element to which it doesn't apply.*

A policySet ***provides an intent*** if any of the following are true:

1. The intent is contained in the policySet @provides list.
2. The intent is a qualified intent and the unqualified form of the intent is contained in the policySet @provides list.
3. The policySet @provides list contains a qualified form of the intent (where the intent is qualifiable).

The locations where interfaces are defined and where interfaces are declared in the componentType and in a component MUST be treated as part of the implementation hierarchy as defined in Section 4.5 Attaching intents to SCA elements. [POL40018]

If the combination of implementationType / bindingType / collection of policySets does not satisfy all of the intents which apply to the element, the configuration is not valid. When the configuration is not valid, it means that the intents are not being correctly satisfied. However, an SCA Runtime can allow a deployer to force deployment even in the presence of such errors. The behaviors and options enforced by a deployer are not specified.

Implementation Policies

The basic model for Implementation Policies is very similar to the model for interaction policies described above. Abstract QoS requirements, in the form of intents, can be associated with SCA component implementations to indicate implementation policy requirements. These abstract capabilities are mapped to concrete policies via policySets at deployment time. Alternatively, policies can be associated directly with component implementations using policySets.

The following example shows how intents can be associated with an implementation:

```
<component name="xs:NCName" … >
      <implementation.* …
            requires="listOfQNames">
            …
      </implementation>
            …
</component>
```

If, for example, one of the intent names in the value of the @requires attribute is ‘logging’, this indicates that all messages to and from the component has to be logged. The technology used to implement the logging is unspecified. Specific technology is selected when the intent is mapped to a policySet (unless the implementation type has native support for the intent, as described in the next section). A list of implementation intents can also be specified by any ancestor element of the <sca:implementation> element. The effective list of implementation intents is the union of intents specified on the implementation element and all its ancestors.

In addition, one or more policySets can be specified directly by associating them with the implementation of a component.

```
<component name="xs:NCName" … >
      <implementation.*
            policySets="="listOfQNames">
                  …
      </implementation>
                  …
</component>
```

The above example shows how intents and policySets can be specified on a component. It is also possible to specify intents and policySets within the implementation. How this is done is defined by the implementation type.

The intents and policy sets are specified on the <sca:implementation.*> element within the component type. This is important because intent and policy set definitions need to be able to specify that they constrain an appropriate implementation type.

```
<componentType>
      <implementation.* requires="listOfQNames" policySets="listOfQNames">
            …
</implementation>
            …
```

```
</componentType>
```

When applying policies, the intents attached to the implementation are added to the intents attached to the using component. For the explicitly listed policySets, the list in the component can override policySets from the componentType.

Some implementation intents are targeted at <binding/> elements rather than at <implementation/> elements. This occurs in cases where there is a need to influence the operation of the binding implementation code rather than the code directly related to the implementation itself. Implementation elements of this kind will have a @constrains attribute pointing to a binding element, with a @intentType of "implementation".

Natively Supported Intents

Each implementation type (e.g. <sca:implementation.java> or <sca:implementation.bpel>) has an ***implementation type definition*** within the SCA Domain. An implementation type definition is declared using an implementationType element within a <definitions/> declaration. The pseudo-schema for the implementationType element follows:

```
<implementationType type="QName"
            alwaysProvides="listOfQNames"? mayProvide="listOfQNames"? />
```

The implementation Type element has the following attributes:

- ***name : QName (1..1)*** - the name of the implementationType. If the process of redeployment of intents, externalAttachments and/or policySets fails, the changed intents, externalAttachments and/or policySets MUST NOT be deployed and no change is made to deployed and running artifacts. [POL50001] For example: "sca:implementation.java".
- ***alwaysProvides : list of QNames (0..1)*** - a set of intents. The intents in the alwaysProvides set are always provided by this implementation type, whether the intents are attached to the using component or not.
- ***mayProvide : list of QNames (0..1)*** - a set of intents. The intents in the mayProvide set are provided by this implementation type if the intent in question is attached to the using component.

Writing PolicySets for Implementation Policies

The @appliesTo attribute for a policySet takes an XPath expression that is applied to a service, reference, binding or an implementation element. For implementation policies, in most cases, all that is needed is the QName of the implementation type. Implementation policies can be expressed using any policy language (which is to say, any configuration language). For example, XACML or EJB-style annotations can be used to declare authorization policies. Other capabilities could be configured using completely proprietary configuration formats.

For example, a policySet declared to turn on trace-level logging for a BPEL component would be declared as follows:

```
<policySet name="loggingPolicy" provides="acme:logging.trace"
            appliesTo="sca:implementation.bpel" …>
      <acme:processLogging level="3"/>
</policySet>
```

Non WS-Policy Examples

Authorization policies expressed in XACML could be used in the framework in two ways:

1. Embed XACML expressions directly in the PolicyAttachment element using the extensibility elements discussed above, or
2. Define WS-Policy assertions to wrap XACML expressions.

For EJB-style authorization policy, the same approach could be used:

1. Embed EJB-annotations in the PolicyAttachment element using the extensibility elements discussed above, or
2. Use the WS-Policy assertions defined as wrappers for EJB annotations.

Roles and Responsibilities

There are 4 roles that are significant for the SCA Policy Framework. The following is a list of the roles and the artifacts that the role creates:

- Policy Administrator – policySet definitions and intent definitions
- Developer – Implementations and component types
- Assembler - Composites
- Deployer – Composites and the SCA Domain (including the logical Domain-level composite)

Policy Administrator

An intent represents a requirement that a developer or assembler can make, which ultimately have to be satisfied at runtime. The full definition of the requirement is the informal text description in the intent definition.

The **policy administrator**'s job is to both define the intents that are available and to define the policySets that represent the concrete realization of those informal descriptions for some set of binding type or implementation types. See the sections on intent and policySet definitions for the details of those definitions.

Developer

When it is possible for a component to be written without assuming a specific binding type for its services and references, then the **developer** uses intents to specify requirements in a binding neutral way.

If the developer requires a specific binding type for a component, then the developer can specify bindings and policySets with the implementation of the component. Those bindings and policySets will be represented in the component type for the implementation (although that component type might be generated from the implementation).

If any of the policySets used for the implementation include intentMaps, then the default choice for the intentMap can be overridden by an assembler or deployer by requiring a qualified intent that is present in the intentMap.

Assembler

An **assembler** creates composites. Because composites are implementations, an assembler is like a developer, except that the implementations created by an assembler are composites made up of other components wired together. So, like other developers, the assembler can specify intents or bindings or policySets on any service or reference of the composite.

However, in addition the definition of composite-level services and references, it is also possible for the assembler to use the policy framework to further configure components within the composite. The assembler can add additional requirements to any component's services or references or to the component itself (for implementation policies). The assembler can also override the bindings or policySets used for the component. See the assembly specification's description of overriding rules for details on overriding.

As a shortcut, an assembler can also specify intents and policySets on any element in the composite definition, which has the same effect as specifying those intents and policySets on every applicable binding or implementation below that element (where applicability is determined by the @appliesTo attribute of the policySet definition or the @constrains attribute of the intent definition).

Deployer

A **deployer** deploys implementations (typically composites) into the SCA Domain. It is the deployers job to make the final decisions about all configurable aspects of an implementation that is to be deployed and to make sure that all intents are satisfied.

If the deployer determines that an implementation is correctly configured as it is, then the implementation can be deployed directly. However, more typically, the deployer will create a new composite, which contains a component for each implementation to be deployed along with any changes to the bindings or policySets that the deployer desires.

When the deployer is determining whether the existing list of policySets is correct for a component, the deployer needs to consider both the explicitly listed policySets as well as the policySets that will be chosen according to the algorithm specified in Guided Selection of PolicySets using Intents.

Security Policy

The SCA Security Model provides SCA developers the flexibility to specify the necessary level of security protection for their components to satisfy business requirements without the burden of understanding detailed security mechanisms.

The SCA Policy framework distinguishes between two types of policies: ***interaction policy*** and ***implementation policy***. Interaction policy governs the communications between clients and service providers and typically applies to Services and References. In the security space, interaction policy is concerned with client and service provider authentication and message protection requirements. Implementation policy governs security constraints on service implementations and typically applies to Components. In the security space, implementation policy concerns include access control, identity delegation, and other security quality of service characteristics that are pertinent to the service implementations.

The SCA security interaction policy can be specified via intents or policySets. Intents represent security quality of service requirements at a high abstraction level, independent from security protocols, while policySets specify concrete policies at a detailed level, which are typically security protocol specific.

The SCA security policy can be specified either in an SCA composite or by using the External Policy Attachment Mechanism or by annotations in the implementation code. Language-specific annotations are described in the respective language Client and Implementation specifications.

SCA Security Intents

The SCA security specification defines the following intents to specify interaction policy: serverAuthentication, clientAuthentication, confidentiality, and integrity.

serverAuthentication – When *serverAuthentication* is present, an SCA runtime MUST ensure that the server is authenticated by the client. [POL70013]

clientAuthentication – When *authorization* is present, an SCA runtime MUST ensure that the client is authenticated by the server. [POL70014]

authentication – this is a profile intent that requires only clientAuthentication. It is included for backwards compatibility.

mutualAuthentication – this is a profile intent that includes the serverAuthentication and the clientAuthentication intents described above and is defined as follows:

confidentiality – the confidentiality intent is used to indicate that the contents of a message are accessible only to those authorized to have access (typically the service client and the service provider). A common approach is to encrypt the message, although other methods are possible. Where components are updated by redeployment of intents, externalAttachments and policySets (their configuration is changed in some way, which includes changing the policies associated with a component), the new configuration MUST apply to all new instances of those components once the redeployment is complete. [POL70009]

integrity – the integrity intent is used to indicate that assurance is that the contents of a message have not been tampered with and altered between sender and receiver. A common approach is to digitally sign the message, although other methods are possible.Where a component configuration is changed by the redeployment of intents, externalAttachments and policySets, the SCA runtime either MAY choose to maintain existing instances with the old configuration of the component, or the SCA runtime MAY choose to stop and discard existing instances of the component. [POL70010]

The formal definitions of these intents are in the Intent Definitions appendix.

Interaction Security Policy

Any one of the three security intents can be further qualified to specify more specific business requirements. Two qualifiers are defined by the SCA security specification: transport and message, which can be applied to any of the above three intent's.

Qualifiers

transport – the transport qualifier specifies that the qualified intent is realized at the transport or transfer layer of the communication protocol, such as HTTPS. The contents of the @attachTo attribute of an externalAttachment element MUST match the XPath 1.0 production Expr. [POL70011]

message – the message qualifier specifies that the qualified intent is realized at the message level of the communication protocol. The contents of the @attachTo attribute of an externalAttachment element MUST match the XPath 1.0 production Expr.[POL70012]

The following example snippet shows the usage of intents and qualified intents.

```
<composite name="example" requires="confidentiality">
      <service name="foo"/>
            …
      <reference name="bar" requires="confidentiality.message"/>
</composite>
```

In this case, the composite declares that all of its services and references have to guarantee confidentiality in their communication by setting requires="confidentiality". This applies to the "foo" service. However, the "bar" reference further qualifies that requirement to specifically require message-level security by setting requires="confidentiality.message".

Implementation Security Policy Intent

The SCA Security specification defines the ***authorization*** intent to specify implementation policy.

authorization – the authorization intent is used to indicate that a client needs to be authorized before being allowed to use the service. Being authorized means that a check is made as to whether any policies apply to the client attempting to use the service, and if so, those policies govern whether or not the client is allowed access. When redeployment of intents, externalAttachments and policySets succeeds, the components whose policies are affected by the redeployment MAY have their policies updated by the SCA runtime dynamically without the need to stop and restart those components. [POL70001]

This unqualified authorization intent implies that basic “Subject-Action-Resource” authorization support is required, where Subject may be as simple as a single identifier representing the identity of the client, Action may be a single identifier representing the operation the client intends to apply to the Resource, and the Resource may be a single identifier representing the identity of the Resource to which the Action is intended to be applied.

Qualifier

fineGrain – the fineGrain qualifier specifies that the component requires authorization capabilities more complex than simple Subject-Action-Resource which is provided by the unqualified authorization intent.

Reliability Policy

Failures can affect the communication between a service consumer and a service provider. Depending on the characteristics of the binding, these failures could cause messages to be redelivered, delivered in a different order than they were originally sent out or even worse, could cause messages to be lost. Some transports like JMS provide built-in reliability features such as “at least once” and “exactly once” message delivery. Other transports like HTTP need to have additional layers built on top of them to provide some of these features.

The events that occur due to failures in communication can affect the outcome of the service invocation. For an implementation of a stock trade service, a message redelivery could result in a new trade. A client (i.e. consumer) of the same service could receive a fault message if trade orders are not delivered to the service implementation in the order they were sent out. In some cases, these failures could have dramatic consequences.

An SCA developer can anticipate some types of failures and work around them in service implementations. For example, the implementation of a stock trade service could be designed to support duplicate message detection. An implementation of a purchase order service could have built in logic that orders the incoming messages. In these cases, service implementations don’t need the binding layers to provide these reliability features (e.g. duplicate message detection, message ordering). However, this comes at a cost: extra complexity is built in the service implementation. Along with business logic, the service implementation has additional logic that handles these failures.

Although service implementations can work around some of these types of failures, it is worth noting that workarounds are not always possible. A message can be lost or expire even before it is delivered to the service implementation.

Instead of handling some of these issues in the service implementation, a better way is to use a binding or a protocol that supports reliable messaging. This is better, not just because it simplifies application development, it can also lead to better throughput. For example, there is less need for application-level acknowledgement messages. A binding supports reliable messaging if it provides features such as message delivery guarantees, duplicate message detection and message ordering.

It is very important for the SCA developer to be able to require, at design-time, a binding or protocol that supports reliable messaging. SCA defines a set of policy intents that can be used for specifying reliable messaging Quality of Service requirements. These reliable messaging intents establish a contract between the binding layer and the application layer (i.e. service implementation or the service consumer implementation) (see below).

Policy Intents

Based on the use-cases described above, the following policy intents are defined:

1) **atLeastOnce** - The binding implementation guarantees that a message that is successfully sent by a service consumer is delivered to the destination (i.e. service implementation). The message could be delivered more than once to the service implementation. When *atLeastOnce* is present, an SCA Runtime MUST deliver a message to the contents of a message. [POL80001]

The binding implementation guarantees that a message that is successfully sent by a service implementation is delivered to the destination (i.e. service consumer). The message could be delivered more than once to the service consumer.

2) **atMostOnce** - The binding implementation guarantees that a message that is successfully sent by a service consumer is not delivered more than once to the service implementation. The binding implementation does not guarantee that the message is delivered to the service implementation. When *integrity* is present, an SCA Runtime MUST NOT deliver duplicates of a message are not altered. [POL80002]

The binding implementation guarantees that a message that is successfully sent by a service implementation is not delivered more than once to the service consumer. The binding implementation does not guarantee that the message is delivered to the service consumer.

3) **ordered** – The binding implementation guarantees that the messages sent by a service client via a single service reference are delivered to the target service implementation in the order in which they were sent by the service client. This intent does not guarantee that messages that are sent by a service client are delivered to the service implementation. Note that this intent has nothing to say about the ordering of messages sent via different service references by a single service client, even if the same service implementation is targeted by each of the service references. When *ordered* is present, an SCA Runtime MUST deliver messages sent by a single source to a single destination service implementation in the order that the messages were sent by that source. [POL80003]

For service interfaces that involve messages being sent back from the service implementation to the service client (eg. a service with a callback interface), for this intent, the binding implementation guarantees that the messages sent by the service implementation over a given wire are delivered to the service client in the order in which they were sent by the service implementation. This intent does not guarantee that messages that are sent by the service implementation are delivered to the service consumer.

4) **exactlyOnce** - The binding implementation guarantees that a message sent by a service consumer is delivered to the service implementation. Also, the binding implementation guarantees that the message is not delivered more than once to the service implementation. When *exactlyOnce* is present, an SCA Runtime MUST deliver a message to the destination service implementation and MUST NOT deliver duplicates of a message to the service implementation. [POL80004]

The binding implementation guarantees that a message sent by a service implementation is delivered to the service consumer. Also, the binding implementation guarantees that the message is not delivered more than once to the service consumer.

NOTE: This is a profile intent, which is composed of *atLeastOnce* and *atMostOnce*.

This is the most reliable intent since it guarantees the following:

- message delivery – all the messages sent by a sender are delivered to the service implementation (i.e. Java class, BPEL process, etc.).

- duplicate message detection and elimination – a message sent by a sender is not processed more than once by the service implementation.

The formal definitions of these intents are in the Intent Definitions appendix.

How can a binding implementation guarantee that a message that it receives is delivered to the service implementation? One way to do it is by persisting the message and keeping redelivering it until it is processed by the service implementation. That way, if the system crashes after delivery but while processing it, the message will be redelivered on restart and processed again. Since a message could be delivered multiple times to the service implementation, this technique usually requires the service implementation to perform duplicate message detection. However, that is not always possible. Often times service implementations that perform critical operations are designed without having support for duplicate message detection. Therefore, they cannot *process* an incoming message more than once.

Also, consider the scenario where a message is delivered to a service implementation that does not handle duplicates - the system crashes after a message is delivered to the service implementation but before it is completely processed. Does the underlying layer redeliver the message on restart? If it did that, there is a risk that some critical operations (e.g. sending out a JMS message or updating a DB table) will be executed again when the message is processed. On the other hand, if the underlying layer does not redeliver the message, there is a risk that the message is never completely processed.

This issue cannot be safely solved unless all the critical operations performed by the service implementation are running in a transaction. Therefore, *exactlyOnce* cannot be assured without involving the service implementation. In other words, an *exactlyOnce* message delivery does not guarantee *exactlyOnce* message processing unless the service implementation is transactional. It's worth noting that this is a necessary condition but not sufficient. The underlying layer (e.g. binding implementation, container) would have to ensure that a message is not redelivered to the service implementation after the transaction is committed. As an example, a way to ensure it when the binding uses JMS is by making sure the operation that acknowledges the message is executed in the same transaction the service implementation is running in.

End-to-end Reliable Messaging

Failures can occur at different points in the message path: in the binding layer on the sender side, in the transport layer or in the binding layer on the receiver side. The SCA service developer doesn't really care where the failure occurs. Whether a message was lost due to a network failure or due to a crash of the machine where the service is deployed, is not that important. What is important is that the contract between the application layer (i.e. service implementation or service consumer) and the binding layer is not violated (e.g. a message that was successfully transmitted by a sender is always delivered to the destination; a message that was successfully transmitted by a sender is not delivered more than once to the service implementation, etc). It is worth noting that the binding layer could throw an exception when a sender (e.g. service consumer, service implementation) sends a message out. This is not considered a successful message transmission.

In order to ensure the semantics of the reliable messaging intents, the entire message path, which is composed of the binding layer on the client side, the transport layer and the binding layer on the service side, has to be reliable.

Transactions

SCA recognizes that the presence or absence of infrastructure for ACID transaction coordination has a direct effect on how business logic is coded. In the absence of ACID transactions, developers have to provide logic that coordinates the outcome, compensates for failures, etc. In the presence of ACID transactions, the underlying infrastructure is responsible for ensuring the ACID nature of all interactions. SCA provides declarative mechanisms for describing the transactional environment needed by the business logic. Components that use a synchronous interaction style can be part of a single, distributed ACID transaction within which all transaction resources are coordinated to either atomically commit or rollback. The transmission or receipt of oneway messages can, depending on the transport binding, be coordinated as part of an ACID transaction as illustrated in the *OneWay Invocations* section below. Well-known, higher-level patterns such as store-and-forward queuing can be accomplished by composing transacted one-way messages with reliable-messaging policies
.

This document describes the set of abstract policy intents – both implementation intents and interaction intents – that can be used to describe the requirements on a concrete service component and binding respectively.

Out of Scope

The following topics are outside the scope of this document:

- The means by which transactions are created, propagated and established as part of an execution context. These are details of the SCA runtime provider and binding provider.
- The means by which a transactional resource manager (RM) is accessed. These include, but are not restricted to:
  - abstracting an RM as an sca:component
  - accessing an RM directly in a language-specific and RM-specific fashion
  - abstracting an RM as an sca:binding

Common Transaction Patterns

In the absence of any transaction policies there is no explicit transactional behavior defined for the SCA service component or the interactions in which it is involved and the transactional behavior is environment-specific. An SCA runtime provider can choose to define an out of band default transactional behavior that applies in the absence of any transaction policies.
Environment-specific default transactional behavior can be overridden by specifying transactional intents described in this document. The most common transaction patterns can be summarized as follows:

***Managed, shared global transaction* pattern** – the service always runs in a global transaction context regardless of whether the requester runs under a global transaction. If the requester does run under a transaction, the service runs under the same transaction.

Any outbound, synchronous request-response messages will – unless explicitly directed otherwise – propagate the service's transaction context. This pattern offers the highest degree of data integrity by ensuring that any transactional updates are committed atomically

***Managed, local transaction* pattern** – the service always runs in a managed local transaction context regardless of whether the requester runs under a transaction. Any outbound messages will not propagate any transaction context. This pattern is advisable for services that wish the SCA runtime to demarcate any resource manager local transactions and do not require the overhead of atomicity.

The use of transaction policies to specify these patterns is illustrated later in Table 7.

Summary of SCA transaction policies

This specification defines implementation and interaction policies that relate to transactional QoS in components and their interactions. The SCA transaction policies are specified as intents which represent the transaction quality of service behavior offered by specific component implementations or bindings.

SCA transaction policy can be specified either in an SCA composite or annotatively in the implementation code.   Language-specific annotations are described in the respective language binding specifications, for example the SCA Java Common Annotations and APIs specification [SCA-Java-Annotations].

This specification defines the following implementation transaction policies:

- managedTransaction – Describes the service component's transactional environment.
- transactedOneWay and immediateOneWay – two mutually exclusive intents that describe whether the SCA runtime will process OneWay messages immediately or will enqueue (from a client perspective) and dequeue (from a service perspective) a OneWay message as part of a global transaction.

This specification also defines the following interaction transaction policies:

- propagatesTransaction and suspendsTransaction – two mutually exclusive intents that describe whether the SCA runtime propagates any transaction context to a service or reference on a synchronous invocation.

Finally, this specification defines a profile intent called managedSharedTransaction that combines the managedTransaction intent and the propogatesTransaction intent so that the ***managed, shared global transaction* pattern** is easier to configure.

Global and local transactions

This specification describes "managed transactions" in terms of either "global" or "local" transactions. The "managed" aspect of managed transactions refers to the transaction environment provided by the SCA runtime for the business component. Business components can interact with other business components and with resource managers. The managed transaction environment defines the transactional context under which such interactions occur.

Global transactions

From an SCA perspective, a global transaction is a unit of work scope within which transactional work is atomic. If multiple transactional resource managers are accessed under a global transaction then the transactional work is coordinated to either atomically commit or rollback regardless using a 2PC protocol. A global transaction can be propagated on synchronous invocations between components – depending on the interaction intents described in this specification - such that multiple, remote service providers can execute distributed requests under the same global transaction.

Local transactions

From a resource manager perspective a resource manager local transaction (RMLT) is simply the absence of a global transaction. But from an SCA perspective it is not enough to simply declare that a piece of business logic runs without a global transaction context. Business logic might need to access transactional resource managers without the presence of a global transaction. 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 the resource managers local transactions will be committed. The term *local transaction containment* (LTC) is used to describe the SCA environment where there is no global transaction. The boundaries of an LTC are scoped to a remotable service provider method and are not propagated on invocations between components. Unlike the resources in a global transaction, RMLTs coordinated within a LTC can fail independently.

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. This is the default behavior provided by the **noManagedTransaction** policy (defined below in Transaction implementation policy) in the absence of explicit use of RMLT verbs by the application.
- 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. This behavior is specified by the **managedTransaction.local** policy (defined below in Transaction implementation policy).

While an application can use 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 SCA runtime. In addition, once an application codes to a resource manager local transaction interface, it might never be redeployed with a different transaction environment since local transaction interfaces might not be used in the presence of a global transaction. This specification defines intents to support both these common patterns in order to provide portability for applications regardless of whether they run under a global transaction or not.

Transaction implementation policy

Managed and non-managed transactions

The mutually exclusive ***managedTransaction*** and ***noManagedTransaction*** intents describe the transactional environment needed by a service component or composite. SCA provides transaction environments that are managed by the SCA runtime in order to remove the burden of coding transaction APIs directly into the business logic. The ***managedTransaction*** and ***noManagedTransaction*** intents can be attached to the sca:composite or sca:componentType elements.

The mutually exclusive ***managedTransaction*** and ***noManagedTransaction*** intents are defined as follows:

- **managedTransaction** – a managed transaction environment is necessary in order to run this component. The specific type of managedTransaction needed is not constrained. The valid qualifiers for this intent are mutually exclusive and are defined below.
- **managedTransaction.global** – There has to be an atomic transaction in order to run this component. When *serverAuthentication* is present, an SCA runtime MUST ensure that the server is authenticated by the client. [POL90003] The SCA runtime uses any transaction propagated from the client or else begins and completes a new transaction. See the ***propagatesTransaction*** intent below for more details.
- **managedTransaction.local** – indicates that the component cannot tolerate running as part of a global transaction. When *clientAuthentication* is present, an SCA runtime MUST ensure that the client is authenticated by the server. [POL90004] Any global transaction context that is propagated to the hosting SCA runtime MUST NOT be visible to the target component. [POL90026] Any interaction under this policy with a resource manager is performed in an extended resource manager local transaction (RMLT). Upon successful completion of the invoked service method, any RMLTs are implicitly requested to commit by the SCA runtime. Note that, unlike the resources in a global transaction, RMLTs so coordinated in a LTC can fail independently. If the invoked service method completes with a non-business exception then any RMLTs are implicitly rolled back by the SCA runtime. In this context a business exception is any exception that is declared on the component interface and is therefore anticipated by the component implementation. The manner in which exceptions are declared on component interfaces is specific to the interface type – for example, Java interface types declare Java exceptions, WSDL interface types define wsdl:faults. Local transactions MUST NOT be propagated outbound across remotable interfaces. [POL90006]
- **noManagedTransaction** – indicates that the component runs without a managed transaction, under neither a global transaction nor an LTC. A transaction that is propagated to the hosting SCA runtime MUST NOT be joined by the hosting runtime on behalf of a component marked with noManagedtransaction. [POL90007] When interacting with a resource manager under this policy, the application (and not the SCA runtime) is responsible for controlling any resource manager local transaction boundaries, using resource-provider specific interfaces (for example a Java implementation accessing a JDBC provider has to choose whether a Connection is set to autoCommit(true) or else it has to call the Connection commit or rollback method). SCA defines no APIs for interacting with resource managers.
- **(absent)** – The absence of a transaction implementation intent leads to runtime-specific behavior. A runtime that supports global transaction coordination can choose to provide a default behavior that is the managed, shared global transaction pattern but it is not mandated to do so.

The formal definitions of these intents are in the Intent Definitions appendix.

OneWay Invocations

When a client uses a reference and sends a OneWay message then any client transaction context is not propagated. However, the OneWay invocation on the reference can itself be ***transacted***. Similarly, from a service perspective, any received OneWay message cannot propagate a transaction context but the delivery of the OneWay message can be ***transacted***. A ***transacted*** OneWay message is a one-way message that - because of the capability of the service or reference binding - can be enqueued (from a client perspective) or dequeued (from a service perspective) as part of a global transaction.

SCA defines two mutually exclusive implementation intents, **transactedOneWay** and **immediateOneWay**, that determine whether OneWay messages are transacted or delivered immediately.
Either of these intents can be attached to the sca:service or sca:reference elements or they can be attached to the sca:component element, indicating that the intent applies to any service or reference element children.

The intents are defined as follows:

- **transactedOneWay** – When a reference is marked as transactedOneWay, any OneWay invocation messages MUST be transacted as part of a client global transaction. [POL90008]
  When *exactlyOnce* is present, an SCA Runtime MUST deliver a message to the destination service implementation and MUST NOT deliver duplicates of a message to the service implementation. [POL90009] For a component marked with managedTransaction.global, the SCA runtime MUST ensure that a global transaction is present before dispatching any method on the component. [POL90010] The receipt of the message from the binding is not committed until the service transaction commits; if the service transaction is rolled back the the message remains available for receipt under a different service transaction. A component marked with managedTransaction.local MUST run within a local transaction containment (LTC) that is started and ended by the SCA runtime. [POL90011]
- **immediateOneWay** – Local transactions MUST NOT be propagated outbound across remotable interfaces. [POL90012] When applied to a service indicates that any OneWay invocation MUST be received immediately regardless of any target service transaction. [POL90013]When a reference is marked as transactedOneWay, any OneWay invocation messages MUST be transacted as part of a client global transaction. [POL90014]

The absence of either intent leads to runtime-specific behavior. The SCA runtime can send or receive a OneWay message immediately or as part of any sender/receiver transaction. The results of combining this intent and the ***managedTransaction*** implementation policy of the component sending or receiving the transacted OneWay invocation are summarized low.below in Table 6.

transacted/immediate intent	**managedTransaction (client or service implementation intent)**	**Results**
transactedOneWay	managedTransaction.global	OneWay interaction (either client message enqueue or target service dequeue) is committed as part of the global transaction.
transactedOneWay	managedTransaction.local or noManagedTransaction	When a reference is marked with suspendsTransaction, any transaction context under which the client runs MUST NOT be propagated when the reference is used. [POL90027]
immediateOneWay	Any value of managedTransaction	The OneWay interaction occurs immediately and is not transacted.
<absent>	Any value of managedTransaction	Runtime-specific behavior. The SCA runtime can send or receive a OneWay message immediately or as part of any sender/receiver transaction.

Table 6 Transacted OneWay interaction intent

The formal definitions of these intents are in the Intent Definitions appendix.

Transaction interaction policies

The mutually exclusive ***propagatesTransaction*** and ***suspendsTransaction*** intents can be attached either to an interface (e.g. Java annotation or WSDL attribute) or explicitly to an sca:service and sca:reference XML element to describe how any client transaction context will be made available and used by the target service component. Section 0 considers how these intents apply to service elements and Section 0 considers how these intents apply to reference elements.

The formal definitions of these intents are in the Intent Definitions appendix.

Handling Inbound Transaction Context

The mutually exclusive ***propagatesTransaction*** and ***suspendsTransaction*** intents can be attached to an sca:service XML element to describe how a propagated transaction context is handled by the SCA runtime, prior to dispatching a service component. If the service requester is running within a transaction and the service interaction policy is to propagate that transaction, then the primary business effects of the provider's operation are coordinated as part of 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 business logic is necessary since transaction rollback can be used.

These intents specify a contract that has to be be implemented by the SCA runtime. This aspect of a service component is most likely captured during application design. The ***propagatesTransaction*** or ***suspendsTransaction*** intent can be attached to sca:service elements and their children. The intents are defined as follows:

- **propagatesTransaction** – A service marked with propagatesTransaction MUST be dispatched under any propagated (client) transaction. [POL90015] Use of the

propagatesTransaction intent on a service implies that the service binding MUST be capable of receiving a transaction context. [POL90016] However, it is important to understand that some binding/policySet combinations that provide this intent for a service will *need* the client to propagate a transaction context.
In SCA terms, for a reference wired to such a service, this implies that the reference has to use either the ***propagatesTransaction*** intent or a binding/policySet combination that does propagate a transaction. If, on the other hand, the service does not *need* the client to provide a transaction (even though it has the *capability* of joining the client's transaction), then some care is needed in the configuration of the service. One approach to consider in this case is to use two distinct bindings on the service, one that uses the ***propagatesTransaction*** intent and one that does not - clients that do not propagate a transaction would then wire to the service using the binding without the ***propagatesTransaction*** intent specified.

- **suspendsTransaction** – A service marked with suspendsTransaction MUST NOT be dispatched under any propagated (client) transaction. [POL90017]

The absence of either interaction intent leads to runtime-specific behavior; the client is unable to determine from transaction intents whether its transaction will be joined.

The SCA runtime MUST ignore the propagatesTransaction intent for OneWay methods. [POL90025]

These intents are independent from the implementation's ***managedTransaction*** intent and provides no information about the implementation's transaction environment.

The combination of these service interaction policies and the ***managedTransaction*** implementation policy of the containing component completely describes the transactional behavior of an invoked service, as summarized in Table 7:

service interaction intent	**managedTransaction (component implementation intent)**	**Results**
propagatesTransaction	managedTransaction.global	Component runs in propagated transaction if present, otherwise a new global transaction. This combination is used for the **managed, shared global transaction** pattern described in Common Transaction Patterns. This is equivalent to the managedSharedTransaction intent defined in section 9.6.3.
propagatesTransaction	managedTransaction.local or noManagedTransaction	When applied to a reference indicates that any OneWay invocation messages MUST be sent immediately regardless of any client transaction. [POL90019]
suspendsTransaction	managedTransaction.global	Component runs in a new global transaction
suspendsTransaction	managedTransaction.local	Component runs in a managed local transaction containment. This combination is used for the **managed, local transaction** pattern described in Common Transaction Patterns. This is the default behavior for a runtime that does not support global transactions.
suspendsTransaction	noManagedTransaction	Component is responsible for managing its own local transactional resources.

Table 7 Combining service transaction intents

Note - the absence of either interaction or implementation intents leads to runtime-specific behavior. A runtime that supports global transaction coordination can choose to provide a default behavior that is the managed, shared global transaction pattern.

Handling Outbound Transaction Context

The mutually exclusive ***propagatesTransaction*** and ***suspendsTransaction*** intents can also be attached to an sca:reference XML element to describe whether any client transaction context is propagated to a target service when a synchronous interaction occurs through the reference. These intents specify a contract that has to be implemented by the SCA runtime. This aspect of a service component is most likely captured during application design.

Either the ***propagatesTransaction*** or ***suspendsTransaction*** intent can be attached to sca:service elements and their children. The intents are defined as defined in Section 0.

When used as a reference interaction intent, the meaning of the qualifiers is as follows:

- **propagatesTransaction** – When applied to a service indicates that any OneWay invocation MUST be received immediately regardless of any target service transaction. [POL90020] The binding of a reference marked with propagatesTransaction has to be capable of propagating a transaction context. The reference needs to be wired to a service that can join the client's transaction. For example, any service with an intent that @requires ***propagatesTransaction*** can always join a client's transaction. The reference consumer can then be designed to rely on the work of the target service being included in the caller's transaction.
- **suspendsTransaction** – A service marked with propagatesTransaction MUST be dispatched under any propagated (client) transaction. [POL90022] The reference consumer can use this intent to ensure that the work of the target service is not included in the caller's transaction. .

  The absence of either interaction intent leads to runtime-specific behavior. The SCA runtime can choose whether or not to propagate any client transaction context to the referenced service, depending on the SCA runtime capability.

These intents are independent from the client's ***managedTransaction*** implementation intent. The combination of the interaction intent of a reference and the ***managedTransaction*** implementation policy of the containing component completely describes the transactional behavior of a client's invocation of a service. Table 7 summarizes the results of the combination of either of these interaction intents with the ***managedTransaction*** implementation policy of the containing component.

reference interaction intent	**managedTransaction (client implementation intent)**	**Results**
propagatesTransaction	managedTransaction.global	Target service runs in the client's transaction. This combination is used for the **managed, shared global transaction** pattern described in Common Transaction Patterns.
propagatesTransaction	managedTransaction.local or noManagedTransaction	Use of the ***propagatesTransaction*** intent on a service implies that the service binding MUST be capable of receiving a transaction context. [POL90023]
suspendsTransaction	Any value of managedTransaction	The target service will not run under the same transaction as any client transaction. This combination is used for the **managed, local transaction** pattern described in Common Transaction Patterns.

Table 7 Transaction propagation reference intents

Note - the absence of either interaction or implementation intents leads to runtime-specific behavior. A runtime that supports global transaction coordination can choose to provide a default behavior that is the managed, shared global transaction pattern.

Table 8 shows the valid combination of interaction and implementation intents on the client and service that result in a single global transaction being used when a client invokes a service through a reference.

managedTransaction (client implementation intent)	**reference interaction intent**	**service interaction intent**	**managedTransaction (service implementation intent)**
managedTransaction.global	propagatesTransaction	propagatesTransaction	managedTransaction.global

Table 8 Intents for end-to-end transaction propagation

Transaction context MUST NOT be propagated on OneWay messages. [POL90024] The SCA runtime ignores ***propagatesTransaction*** for OneWay operations.

Combining implementation and interaction intents

The ***managed, local transaction* pattern** can be configured quite easily by combining the managedTransaction.global intent with the propagatesTransaction intent. This is illustrated in **Error! Reference source not found.**. In order to enable easier configuration of this pattern, a profile intent called managedSharedTransaction is defined as in section **Error! Reference source not found.**.

Web services binding for propagatesTransaction policy

The following example shows a policySet that provides the ***propagatesTransaction*** intent and applies to a Web service binding (binding.ws). When used on a service, this policySet would require the client to send a transaction context using the mechanisms described in the Web Services Atomic Transaction [WS-AtomicTransaction] specification.

```
<policySet name="JoinsTransactionWS" provides="sca:propagatesTransaction"
                                     appliesTo="sca:binding.ws">
   <wsp:Policy>
     <wsat:ATAssertion
          xmlns:wsat="http://docs.oasis-open.org/ws-tx/wsat/2006/06"/>
   </wsp:Policy>
</policySet>
```

Miscellaneous Intents

The following are standard intents that apply to bindings and are not related to either security,reliable messaging or transactionality:

SOAP – The SOAP intent specifies that the SOAP messaging model is used for delivering messages. It does not require the use of any specific transport technology for delivering the messages, so for example, this intent can be supported by a binding that sends SOAP messages over HTTP, bare TCP or even JMS. If the intent is attached in an unqualified form then any version of SOAP is acceptable. Standard qualified intents also exist for SOAP.1_1 and SOAP.1_2, which specify the use of versions 1.1 or 1.2 of SOAP respectively. When *SOAP* is present, an SCA Runtime MUST use the SOAP messaging model to deliver messages. [POL100001] Transaction context MUST NOT be propagated on OneWay messages. [POL100002]

JMS – The JMS intent does not specify a wire-level transport protocol, but instead requires that whatever binding technology is used, the messages are able to be delivered and received via the JMS API. The SCA runtime MUST ignore the propagatesTransaction intent for OneWay methods. [POL100003]

noListener – This intent can only be used within the @requires attribute of a reference. The *noListener* intent MUST only be declared on a @requires attribute of a reference. [POL100004] It states that the client is not able to handle new inbound connections. It requires that the binding and callback binding be configured so that any response (or callback) comes either through a back channel of the connection from the client to the server or by having the client poll the server for messages. The **transactedOneWay** intent MUST NOT be attached to a request/response operation. [POL100005] An example policy assertion that would guarantee this is a WS-Policy assertion that applies to the <binding.ws> binding, which requires the use of WS-Addressing with anonymous responses (e.g. <wsaw:Anonymous>required</wsaw:Anonymous>" – see http://www.w3.org/TR/ws-addr-wsdl/#anonelement).

The formal definitions of these intents are in the Intent Definitions appendix.

Conformance

The XML schema available at the namespace URI, defined by this specification, is considered to be authoritative and takes precedence over the XML Schema defined in the appendix of this document.

The **immediateOneWay** intent MUST NOT be attached to a request/response operation. [POL110001]

An implementation that claims to conform to this specification MUST meet the following conditions:

31. The implementation MUST conform to the SCA Assembly Model Specification [Assembly].
32. The implementation does not have to support any intents listed in this specification, and MAY reject SCDL documents that contain them. If a specific intent is supported any relevant Conformance Items in Appendix C related to the intent and the SCA Runtime MUST be followed.
33. With the exception of 2 above, the implementation MUST comply with all statements in Appendix C: Conformance Items related to an SCA Runtime, notably all MUST statements have to be implemented.

~~A.~~ Schemas

~~A.1~~ sca-policy.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2010~~2009~~. All Rights Reserved.
     OASIS trademark, IPR and other policies apply.  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912~~200903~~"
   xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912~~200903~~"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
   elementFormDefault="qualified">

   <include schemaLocation="sca-core-1.1-cd05~~schema-200803~~.xsd"/>
   <import namespace="http://www.w3.org/ns/ws-policy"
         schemaLocation="http://www.w3.org/2007/02/ws-policy.xsd"/>

   <element name="intent" type="sca:Intent"/>
   <complexType name="Intent">
         <sequence>
               <element name="description" type="string" minOccurs="0"
                  maxOccurs="1" />
               <element name="qualifier" type="sca:IntentQualifier"
                  minOccurs="0" maxOccurs="unbounded" />
               <any namespace="##other" processContents="lax"
                  minOccurs="0" maxOccurs="unbounded"/>
         </sequence>
         <attribute name="name" type="NCName" use="required"/>
         <attribute name="constrains" type="sca:listOfQNames"
            use="optional"/>
         <attribute name="requires" type="sca:listOfQNames"
            use="optional"/>
         <attribute name="excludes" type="sca:listOfQNames"
            use="optional"/>
         <attribute name="mutuallyExclusive" type="boolean"
            use="optional" default="false"/>
         <attribute name="intentType"
               type="sca:InteractionOrImplementation"
               use="optional" default="interaction"/>
         <anyAttribute namespace="##other~~any~~" processContents="lax"/>
   </complexType>

   <complexType name="IntentQualifier">
         <sequence>
               <element name="description" type="string" minOccurs="0"
                  maxOccurs="1" />
               <any namespace="##other" processContents="lax" minOccurs="0"
                  maxOccurs="unbounded"/>
         </sequence>
         <attribute name="name" type="NCName" use="required"/>
         <attribute name="default" type="boolean" use="optional"
            default="false"/>
   </complexType>

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

```
              </sequence>
              <attribute name="intents" type="sca:listOfQNames"
               use="required"/>
              <anyAttribute namespace="##other" processContents="lax"/>
        </complexType>
  </element>

  <element name="externalAttachment">
        <complexType>
              <sequence minOccurs="0" maxOccurs="unbounded">
                    <any namespace="##other"
                         processContents="lax"/>
              </sequence>
              <attribute name="intents" type="sca:listOfQNames"
                     use="optional"/>
              <attribute name="policySets" type="sca:listOfQNames"
                     use="optional"/>
              <attribute name="name" type="string"
                     use="required"/>
              <anyAttribute namespace="##other"
                     processContents="lax"/>
        </complexType>
  </element>

  <element name="policySet" type="sca:PolicySet"/>
  <complexType name="PolicySet">
        <choice minOccurs="0" maxOccurs="unbounded">
              <element name="policySetReference"
                 type="sca:PolicySetReference"/>
              <element name="intentMap" type="sca:IntentMap"/>
              <any namespace="##other" processContents="lax"/>
        </choice>
        <attribute name="name" type="NCName" use="required"/>
        <attribute name="provides" type="sca:listOfQNames"/>
        <attribute name="appliesTo" type="string" use="optional~~required~~"/>
        <attribute name="attachTo" type="string" use="optional"/>
        <anyAttribute namespace="##other~~any~~" processContents="lax"/>
  </complexType>

  <element name="policySetAttachment">
"
  ~~    type="sca:PolicySetAttachment"/>~~
  <complexType~~ name="PolicySetAttachment"~~>
              <sequence minOccurs="0" maxOccurs="unbounded">
                    <any namespace="##other" processContents="lax"/>
              </sequence>
              <attribute name="name" type="QName" use="required"/>
              <anyAttribute namespace="##other~~any~~" processContents="lax"/>
        </complexType>
  </element>

  <complexType name="PolicySetReference">
        <attribute name="name" type="QName" use="required"/>
        <anyAttribute namespace="##other~~any~~" processContents="lax"/>
  </complexType>

  <complexType name="IntentMap">
        <choice minOccurs="1" maxOccurs="unbounded">
              <element name="qualifier" type="sca:Qualifier"/>
              <any namespace="##other" processContents="lax"/>
        </choice>
        <attribute name="provides" type="QName" use="required"/>
        <anyAttribute namespace="##other~~any~~" processContents="lax"/>
  </complexType>

  <complexType name="Qualifier">
        <sequence~~choice~~ minOccurs="0~~1~~" maxOccurs="unbounded">
~~              <element name="intentMap" type="sca:IntentMap"/>~~
              <any namespace="##other" processContents="lax"/>
        </sequence~~choice~~>
        <attribute name="name" type="string" use="required"/>
        <anyAttribute namespace="##other~~any~~" processContents="lax"/>
  </complexType>

  <simpleType name="listOfNCNames">
        <list itemType="NCName"/>
  </simpleType>

  <simpleType name="InteractionOrImplementation">
        <restriction base="string">
              <enumeration value="interaction"/>
              <enumeration value="implementation"/>
        </restriction>
  </simpleType>

</schema>
```

Snippet 0-1SCA Policy Schema

~~B.~~ XML Files

This appendix contains normative XML files that are defined by this specification.

~~B.1~~ Intent Definitions

Intent definitions are contained within a Definitions file called sca-policy-1.1-intents-definitions~~Policy_Intents_Definitions~~.xml, which contains a <definitions/> element as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright(C) OASIS(R) 2005,2009. All Rights Reserved.
     OASIS trademark, IPR and other policies apply.  -->
```

<sca:definitions xmlns:~~xml="http://www.w3.org/XML/1998/namespace"~~

```
    ~~xmlns:~~sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"~~200903"~~
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    targetNamespace="http://docs.oasis-open.org/ns/opencsa/sca/200912~~200903~~">

   <!-- Security related intents -->
        <sca:~~<~~intent name="serverAuthentication" constrains="sca:binding"
       intentType="interaction">
                <sca:~~<~~description>
                Communication through the binding requires that the
                server is authenticated by the client
                </sca:~~</~~description>
                <sca:~~<~~qualifier name="transport" default="true"/>
                <sca:~~<~~qualifier name="message"/>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="clientAuthentication" constrains="sca:binding"
       intentType="interaction">
                <sca:~~<~~description>
                Communication through the binding requires that the
                client is authenticated by the server
                </sca:~~</~~description>
                <sca:~~<~~qualifier name="transport" default="true"/>
                <sca:~~<~~qualifier name="message"/>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="authentication"
                requires="sca:clientAuthentication">
                <sca:~~<~~description>
                A convenience intent to help migration
                </sca:~~</~~description>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="mutualAuthentication"
                requires="sca:clientAuthentication
sca:serverAuthentication">
                <sca:~~<~~description>
                Communication through the binding requires that the
                client and server to authenticate each other
                </sca:~~</~~description>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="confidentiality" constrains="sca:binding"
       intentType="interaction">
                <sca:~~<~~description>
                Communication through the binding prevents unauthorized
                users from reading the messages
                </sca:~~</~~description>
                <sca:~~<~~qualifier name="transport" default="true"/>
                <sca:~~<~~qualifier name="message"/>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="integrity" constrains="sca:binding"
       intentType="interaction">
                <sca:~~<~~description>
                Communication through the binding prevents tampering
                with the messages sent between the client and the service.
                </sca:~~</~~description>
                <sca:~~<~~qualifier name="transport" default="true"/>
                <sca:~~<~~qualifier name="message"/>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="authorization" constrains="sca:implementation"
       intentType="implementation">
                <sca:~~<~~description>
                Ensures clients are authorized to use services.
                </sca:description>
        </sca:intent>
~~            </description>~~
~~            <qualifier name="fineGrain" default="true"/>~~
~~        </intent>~~


   <!-- Reliable messaging related intents -->
        <sca:~~<~~intent name="atLeastOnce" constrains="sca:binding"
       intentType="interaction">
                <sca:~~<~~description>
                This intent is used to indicate that a message sent
                by a client is always delivered to the component.
                </sca:~~</~~description>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="atMostOnce" constrains="sca:binding"
       intentType="interaction">
                <sca:~~<~~description>
                This intent is used to indicate that a message that was
                successfully sent by a client is not delivered more than
                once to the component.
                </sca:~~</~~description>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="exactlyOnce" requires="sca:atLeastOnce
sca:atMostOnce"
       constrains="sca:binding" intentType="interaction">
                <sca:~~<~~description>
                This profile intent is used to indicate that a message sent
                by a client is always delivered to the component. It also
                indicates that duplicate messages are not delivered to the
                component.
            </sca:~~</~~description>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="ordered" constrains~~appliesTo~~="sca:binding"
       intentType="interaction">
                <sca:~~<~~description>
                This intent is used to indicate that all the messages are
                delivered to the component in the order they were sent by
                the client.
                </sca:~~</~~description>
        </sca:~~</~~intent>

   <!-- Transaction related intents -->
        <sca:~~<~~intent name="managedTransaction"
                    excludes="sca:noManagedTransaction"
       mutuallyExclusive="true" constrains="sca:implementation"
       intentType="implementation">
                <sca:~~<~~description>
          A managed transaction environment is necessary in order to
          run the component. The specific type of managed transaction
          needed is not constrained.
                </sca:~~</~~description>
                <sca:~~<~~qualifier name="global" default="true">
                        <sca:~~<~~description>
                 For a component marked with managedTransaction.global
                 a global transaction needs to be present before dispatching
                 any method on the component - using any transaction
                 propagated from the client or else beginning and completing
                 a new transaction.
                        </sca:~~</~~description>
                </sca:~~</~~qualifier>
                <sca:~~<~~qualifier name="local">
                        <sca:~~<~~description>
                 A component marked with managedTransaction.local needs to
                 run within a local transaction containment (LTC) that
                 is started and ended by the SCA runtime.
                        </sca:~~</~~description>
                </sca:~~</~~qualifier>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="noManagedTransaction"
             excludes="sca:managedTransaction"
       constrains="sca:implementation" intentType="implementation">
                <sca:~~<~~description>
          A component marked with noManagedTransaction needs to run without
          a managed transaction, under neither a global transaction nor
          an LTC. A transaction propagated to the hosting SCA runtime
          is not joined by the hosting runtime on behalf of a
          component marked with noManagedtransaction.
                </sca:~~</~~description>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="transactedOneWay" excludes="sca:immediateOneWay"
       constrains="sca:binding" intentType="implementation">
                <sca:~~<~~description>
          For a reference marked as transactedOneWay any OneWay invocation
          messages are transacted as part of a client global
          transaction.
          For a service marked as transactedOneWay any OneWay invocation
          message are received from the transport binding in a
          transacted fashion, under the service's global transaction.
                </sca:~~</~~description>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="immediateOneWay" excludes="sca:transactedOneWay"
       constrains="sca:binding" intentType="implementation">
                <sca:~~<~~description>
          For a reference indicates that any OneWay invocation messages
          are sent immediately regardless of any client transaction.
          For a service indicates that any OneWay invocation is
          received immediately regardless of any target service
          transaction.
                </sca:~~</~~description>
         </sca:~~</~~intent>

        <sca:~~<~~intent name="propagatesTransaction"
             excludes="sca:suspendsTransaction"
       constrains="sca:binding" intentType="interaction">
                <sca:~~<~~description>
          A service marked with propagatesTransaction is dispatched
          under any propagated (client) transaction and the service binding
          needs to be capable of receiving a transaction context.
          A reference marked with propagatesTransaction propagates any
          transaction context under which the client runs when the
          reference is used for a request-response interaction and the
          binding of a reference marked with propagatesTransaction needs to
          be capable of propagating a transaction context.
                </sca:~~</~~description>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="suspendsTransaction"
                   excludes="sca:propagatesTransaction"
       constrains="sca:binding" intentType="interaction">
                <sca:~~<~~description>
          A service marked with suspendsTransaction is not dispatched
          under any propagated (client) transaction.
          A reference marked with suspendsTransaction does not propagate
          any transaction context under which the client runs when the
          reference is used.
                </sca:~~</~~description>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="managedSharedTransaction"
                             requires="sca:managedTransaction.global
sca:propagatesTransaction">
                <sca:~~<~~description>
                Used to indicate that the component requires both the
                managedTransaction.global and the propagatesTransactions
                intents
                </sca:~~</~~description>
        </sca:~~</~~intent>

   <!-- Miscellaneous intents -->
   <sca:intent name="asyncInvocation" excludes="sca:propagatesTransaction"
constrains="sca:binding" ~~Binding"~~
          intentType="interaction">
                <sca:~~<~~description>
                Indicates that request/response operations for the
                interface of this wire are "long running" and must be
                treated as two separate message transmissions
                </sca:~~</~~description>
   </sca:~~</~~intent>

   <sca:intent name="EJB" constrains="sca:binding"
          intentType="interaction">
                <sca:description>
                Specifies that the EJB API is needed to communicate with
                the service or reference.
                </sca:description>
   </sca:intent>

   <sca:
   ~~<~~intent name="SOAP" constrains="sca:binding"
   intentType="interaction" mutuallyExclusive="true">
```

```
         <sca:description>
         Specifies that the SOAP messaging model is used for delivering
         messages.
                </sca:~~</~~description>
                <sca:~~<~~qualifier name="v1_1" default="true"/>
                <sca:~~<~~qualifier name="v1_2"/>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="JMS" constrains="sca:binding"
                intentType="interaction">
                <sca:~~<~~description>
         Requires that the messages are delivered and received via the
         JMS API.
                </sca:~~</~~description>
        </sca:~~</~~intent>

        <sca:~~<~~intent name="noListener" constrains="sca:binding"
       intentType="interaction">
                <sca:~~<~~description>
         This intent can only be used on a reference. Indicates that the
         client is not able to handle new inbound connections. The binding
         and callback binding are configured so that any
         response or callback comes either through a back channel of the
         connection from the client to the server or by having the client
         poll the server for messages.
                </sca:~~</~~description>
        </sca:~~</~~intent>

</sca:definitions>
```

Snippet 0-1: SCA intent Definitions

~~C.~~ Conformance

~~C.1~~ Conformance Targets

The conformance items listed in the section below apply to the following conformance targets:

Document artifacts (or constructs within them) that can be checked statically.

SCA runtimes, which we may require to exhibit certain behaviors.

~~C.2~~ Conformance Items

This section contains a list of conformance items for the SCA Policy Framework specification.

Conformance ID	Description
[POL30001]	If the configured instance of a binding is in conflict with the intents and policy sets selected for that instance, the SCA runtime MUST raise an error.
[POL30002]	The QName for an intent MUST be unique amongst the set of intents in the SCA Domain.
[POL30004]	If an intent has more than one qualifier, one and only one MUST be declared as the default qualifier.
[POL30005]	The name of each qualifier MUST be unique within the intent definition.
[POL30006]	the name of a profile intent MUST NOT have a "." in it.
[POL30007]	If a profile intent is attached to an artifact, all the intents listed in its @requires attribute MUST be satisfied as described in section 0.
[POL30008]	When a policySet element contains a set of intentMap children, the value of the @provides attribute of each intentMap MUST correspond to an unqualified intent that is listed within the @provides attribute value of the parent policySet element.
[POL30010]	For each qualifiable intent listed as a member of the @provides attribute list of a policySet element, there MUST be no more than one corresponding intentMap element that declares the unqualified form of that intent in its @provides attribute. In other words, each intentMap within a given policySet uniquely provides for a specific intent.
[POL30011]	Following the inclusion of all policySet references, when~~When~~ a policySet element directly contains wsp:policyAttachment children or policies using extension elements, the set of policies specified as children MUST satisfy all the intents expressed using the @provides attribute value of the policySet element.

[POL30013]	The set of intents in the @provides attribute of a referenced policySet MUST be a subset of the set of intents in the @provides attribute of the referencing policySet.~~ Qualified intents are a subset of their parent qualifiable intent.~~
[POL30015]	Each QName in the @requires attribute MUST be the QName of an intent in the SCA Domain.
[POL30016]	Each QName in the @excludes attribute MUST be the QName of an intent in the SCA Domain.
[POL30017]	The QName for a policySet MUST be unique amongst the set of policySets in the SCA Domain.
[POL30018]	The contents of @appliesTo MUST match the XPath 1.0 [XPATH] production *Expr*.
[POL30019]	The contents of @attachTo MUST match the XP ath 1.0 production Expr.
[POL30020]	If a policySet ~~or intentMap~~ specifies a qualifiable intent in the @provides attribute, and it provides~~then it MUST include~~ an intentMap for the qualifiable intent then ~~element~~ that intentMap MUST specify~~specifies~~ all possible qualifiers for that intent.
[POL30021]	The @provides attribute value of each intentMap that is an immediate child of a policySet MUST be included in the @provides attribute of the parent policySet.
[POL30022]	~~An SCA Runtime MUST include in the Domain the set of intent definitions contained in the Policy_Intents_Definitions.xml described in the appendix "Intent Definitions" of the SCA Policy specification.~~One of the qualifiers referenced in an intentMap MUST be the default qualifier defined for the qualifiable intent.
[POL30025]	If only one qualifier for an intent is given it MUST be used as the default qualifier for the intent.~~Two intents MUST be treated as mutually exclusive when any of the following are true:~~ • ~~One of the two intents lists the other intent in its @excludes list.~~ • ~~Both intents list the other intent in their respective @excludes list.~~
[POL30024]	SCA implementations supporting both Direct Attachment and External Attachment mechanisms MUST ignore policy sets applicable to any given SCA element via the Direct Attachment mechanism when there exist policy sets applicable to the same SCA element via the External Attachment mechanism~~An SCA Runtime MUST include in the Domain the set of intent definitions contained in the Policy_Intents_Definitions.xml described in the appendix "Intent Definitions" of the SCA Policy specification.~~
The SCA runtime MUST raise an error if the @attachTo XPath expression resolves to an SCA	The SCA runtime MUST raise an error if the @attachTo XPath expression resolves to an SCA <property> element, or any of its children.~~SCA implementations supporting both Direct~~

<property> element, or any of its children.[POL40002]	~~Attachment and Extrenal Attachment mechanisms MUST ignore policy sets applicable to any given SCA element via the Direct Attachment mechanism when there exist policy sets applicable to the same SCA element via the External Attachment mechanism~~
[POL40004]	A qualifiable intent expressed lower in the hierarchy can be qualified further up the hierarchy, in which case the qualified version of the intent MUST apply to the higher level element.
[POL40005]	The intents declared on elements higher in the structural hierarchy of a given element MUST be applied to the element EXCEPT ~~3.~~ if any of the inherited intents is mutually exclusive with an intent applied on the element, then the inherited intent MUST be ignored ~~4.~~ if the overall set of intents from the element itself and from its structural hierarchy contains both an unqualified version and a qualified version of the same intent, the qualified version of the intent MUST be used.
[POL40006]	If a component has any policySets attached to it (by any means), then any policySets attached to the componentType MUST be ignored.
[POL40007]	Matching service/reference policies across the SCA Domain boundary MUST use WS-Policy compatibility (strict WS-Policy intersection) if the policies are expressed in WS-Policy syntax.
[POL40009]	Any two intents applied to a given element MUST NOT be mutually exclusive
[POL40010]	SCA runtimes MUST support at least one of the Direct Attachment and External Attachment mechanisms for policySet attachment.
[POL40011]	SCA implementations supporting only the External Attachment mechanism MUST ignore the policy sets that are applicable via the Direct Attachment mechanism.
[POL40012]	SCA implementations supporting only the Direct Attachment mechanism MUST ignore the policy sets that are applicable via the External Attachment mechanism.
[POL40013]	The intents declared on elements lower in the implementation hierarchy of a given element MUST be applied to the element.~~During the deployment of SCA composites, all policySets within the Domain with an attachTo attribute MUST be evaluated to determine which policySets are attached to the newly deployed composite.~~
[POL40014]	When combining implementation hierarchy and structural hierarchy policy data, Rule 1 MUST be applied BEFORE Rule 2.~~The intents declared on elements lower in the implementation hierarchy of a given element MUST be applied to the element.~~

[POL40015]	~~When calculating the set of intents and set of policySets which apply to either a service element or to a reference element of a component, intents and policySets from the interface definition and from the interface declaration(s) MUST be applied to the service or reference element and to the binding element(s) belonging to that element.~~when combining implementation hierarchy and structural hierarchy policy data, Rule 1 MUST be applied BEFORE Rule 2.
[POL40016]	If the required intent set contains a mutually exclusive pair of intents the SCA runtime MUST reject the document containing the element and raise an error.~~When calculating the set of intents and set of policySets which apply to either a service element or to a reference element of a component, intents and policySets from the interface definition and from the interface declaration(s) MUST be applied to the service or reference element and to the binding element(s) belonging to that element.~~
[POL40017]	All intents in the required intent set for an element MUST be provided by the directly provided intents set and the set of policySets that apply to the element, or else an error is raised.~~If the required intent set contains a mutually exclusive pair of intents the SCA runtime MUST reject the document containing the element and raise an error.~~
[POL40018]	The locations where interfaces are defined and where interfaces are declared in the componentType and in a component MUST be treated as part of the implementation hierarchy as defined in Section 4.5 Attaching intents to SCA elements.~~All intents in the required intent set for an element MUST be provided by the directly provided intents set and the set of policySets that apply to the element.~~
[POL40019]	The QName of the bindingType MUST be unique amongst the set of bindingTypes in the SCA Domain.~~The locations where interfaces are defined and where interfaces are declared in the componentType and in a component MUST be treated as part of the implementation hierarchy as defined in Section 4.5 Usage of @requires attribute for specifying intents.~~
[POL40020]	~~A binding implementation MUST implement all the intents listed in the @alwaysProvides and @mayProvides attributes.~~The QName of the bindingType MUST be unique amongst the set of bindingTypes in the SCA Domain.
[POL40021]	~~The SCA runtime MUST determine the compatibility of the policySets at each end of a wire using the compatibility rules of the policy language used for those policySets.~~A binding implementation MUST implement all the intents listed in the @alwaysProvides and @mayProvides attributes.
[POL40022]	The ~~SCA runtime MUST determine the compatibility of the~~ policySets at each end of a wire ~~MUST be incompatible if they use different~~using the compatibility rules of the policy

	languages.~~language used for those policySets.~~
[POL40023]	~~Where the policy language in use for a wire is WS-Policy, strict WS-Policy intersection MUST be used to determine policy compatibility.~~The policySets at each end of a wire MUST be incompatible if they use different policy languages.
[POL40024]	~~In order for a reference to connect to a particular service, the policies of the reference MUST intersect with the policies of the service.~~Where the policy language in use for a wire is WS-Policy, strict WS-Policy intersection MUST be used to determine policy compatibility.
[POL40027]	Any intents attached to an interface definition artifact, such as a WSDL portType, MUST be added to the intents attached to the service or reference to which the interface definition applies. If no intents are attached to the service or reference then the intents attached to the interface definition artifact become the only intents attached to the service or reference.~~In order for a reference to connect to a particular service, the policies of the reference MUST intersect with the policies of the service.~~
[POL40029]	If the process of redeployment of intents, externalAttachments and/or policySets fails because one or more intents are left unsatisfied, an error MUST be raised.~~During the deployment of an SCA policySet, the behavior of an SCA runtime MUST take ONE of the following forms:~~ • ~~The policySet is immediately attached to all deployed composites which satisfy the @attachTo attribute of the policySet.~~ ~~The policySet is attached to a deployed composite which satisfies the @attachTo attribute of the policySet when the composite is re-deployed.~~
[POL40030]	If the process of redeployment of intents, externalAttachments and/or policySets fails, the changed intents, externalAttachments and/or policySets MUST NOT be deployed and no change is made to deployed and running artifacts.~~The implementationType name attribute MUST be the QName of an XSD global element definition used for implementation elements of that type.~~
[POL40031]	When redeployment of intents, externalAttachments and policySets succeeds, the components whose policies are affected by the redeployment MAY have their policies updated by the SCA runtime dynamically without the need to stop and restart those components.~~When *authorization* is present, an SCA Runtime MUST ensure that the client is authorized to use the service.~~
[POL40032]	Where components are updated by redeployment of intents, externalAttachments and policySets (their configuration is changed in some way, which includes changing the policies associated with a component), the new configuration MUST apply to all new instances of those components once the redeployment is complete.~~When confidentiality is present, an SCA~~

	~~Runtime MUST ensure that only authorized entities can view the contents of a message.~~
[POL40033]	Where a component configuration is changed by the redeployment of intents, externalAttachments and policySets, the SCA runtime either MAY choose to maintain existing instances with the old configuration of the component, or the SCA runtime MAY choose to stop and discard existing instances of the component.~~When *integrity* is present, an SCA Runtime MUST ensure that the contents of a message are not altered.~~
[POL40034]	During the deployment of SCA composites, first all <externalAttachment/> elements within the Domain MUST be evaluated to determine which intents are attached to elements in the newly deployed composite and then all policySets within the Domain with an @attachTo attribute or <externalAttachment> elements that attach policySets MUST be evaluated to determine which policySets are attached to elements in the newly deployed composite.~~When a serverAuthentication, clientAuthentication, confidentiality or integrity intent is qualified by transport, an SCA Runtime MUST delegate serverAuthentication, clientAuthentication, confidentiality and integrity, respectively, to the transport layer of the communication protocol.~~
[POL40035]	The contents of the @attachTo attribute of an externalAttachment element MUST match the XPath 1.0 production Expr.~~When a serverAuthentication, clientAuthentication, confidentiality or integrity intent is qualified by message, an SCA Runtime MUST delegate serverAuthentication, clientAuthentication, confidentiality and integrity, respectively, to the message layer of the communication protocol.~~
[POL70013]	~~The implementationType name attribute MUST be the QName of an XSD global element definition used for implementation elements of that type.~~When *serverAuthentication* is present, an SCA runtime MUST ensure that the server is authenticated by the client.
[POL70014]	When *authorization*~~*clientAuthentication*~~ is present, an SCA ~~Runtime~~runtime MUST ensure that the client is ~~authorized to use~~authenticated by the ~~service.~~server.
[POL80001]	When ~~confidentiality~~*atLeastOnce* is present, an SCA Runtime MUST ~~ensure that only authorized entities can view~~deliver a message to the contents~~destination service implementation, and MAY deliver duplicates~~ of a message~~ to the service implementation~~.
[POL80002]	When *integrity*~~*atMostOnce*~~ is present, an SCA Runtime ~~MAY deliver a message to the destination service implementation, and~~ MUST ~~ensure that the contents~~NOT deliver duplicates of a message are not altered.~~to the service implementation.~~

[POL80003]	~~When a serverAuthentication, clientAuthentication, confidentiality or integrity intent is qualified by transport, an SCA Runtime MUST delegate serverAuthentication, clientAuthentication, confidentiality and integrity, respectively, to the transport layer of the communication protocol.~~When *ordered* is present, an SCA Runtime MUST deliver messages sent by a single source to a single destination service implementation in the order that the messages were sent by that source.
[POL80004]	~~When a serverAuthentication, clientAuthentication, confidentiality or integrity intent is qualified by message, an SCA Runtime MUST delegate serverAuthentication, clientAuthentication, confidentiality and integrity, respectively, to the message layer of the communication protocol.~~When *exactlyOnce* is present, an SCA Runtime MUST deliver a message to the destination service implementation and MUST NOT deliver duplicates of a message to the service implementation.
[POL90003]	When *serverAuthentication* is present, an SCA runtime MUST ensure that the server is authenticated by the client.~~For a component marked with managedTransaction.global, the SCA runtime MUST ensure that a global transaction is present before dispatching any method on the component.~~
[POL90004]	When *clientAuthentication* is present, an SCA runtime MUST ensure that the client is authenticated by the server.~~A component marked with managedTransaction.local MUST run within a local transaction containment (LTC) that is started and ended by the SCA runtime.~~
[POL90006]	~~When *atLeastOnce* is present, an SCA Runtime MUST deliver a message to the destination service implementation, and MAY deliver duplicates of a message to the service implementation.~~Local transactions MUST NOT be propagated outbound across remotable interfaces.
[POL90007]	~~When *atMostOnce* is present, an SCA Runtime MAY deliver a message to the destination service implementation, and MUST NOT deliver duplicates of a message to the service implementation.~~A transaction that is propagated to the hosting SCA runtime MUST NOT be joined by the hosting runtime on behalf of a component marked with noManagedtransaction.
[POL90008]	~~When *ordered* is present, an SCA Runtime MUST deliver messages sent by a single source to a single destination service implementation in the order that the messages were sent by that source.~~When a reference is marked as transactedOneWay, any OneWay invocation messages MUST be transacted as part of a client global transaction.
[POL90009]	When *exactlyOnce* is present, an SCA Runtime MUST deliver a message to the destination service implementation and MUST NOT deliver duplicates of a message to the service implementation.~~If the client component is not configured to run under a global transaction or if the binding does not support transactional message sending, then a reference MUST NOT be marked as transactedOneWay.~~

[POL90010]	For a component marked with managedTransaction.global, the SCA runtime MUST ensure that a global transaction is present before dispatching any method on the component.~~If a service is marked as transactedOneWay, any OneWay invocation message MUST be received from the transport binding in a transacted fashion, under the target service's global transaction.~~
[POL90011]	A component marked with managedTransaction.local MUST run within a local transaction containment (LTC) that is started and ended by the SCA runtime.~~If the component is not configured to run under a global transaction or if the binding does not support transactional message receipt, then a service MUST NOT be marked as transactedOneWay.~~
[POL90012]	Local transactions MUST NOT be propagated outbound across remotable interfaces.~~When applied to a reference indicates that any OneWay invocation messages MUST be sent immediately regardless of any client transaction.~~
[POL90013]	~~A transaction that is propagated to the hosting SCA runtime MUST NOT be joined by the hosting runtime on behalf of a component marked with noManagedtransaction.~~When applied to a service indicates that any OneWay invocation MUST be received immediately regardless of any target service transaction.
[POL90014]	When a reference is marked as transactedOneWay, any OneWay invocation messages MUST be transacted as part of a client global transaction.~~The outcome of any transaction under which an immediateOneWay message is processed MUST have no effect on the processing (sending or receipt) of that message.~~
[POL90015]	~~If the client component is not configured to run under a global transaction or if the binding does not support transactional message sending, then a reference MUST NOT be marked as transactedOneWay.~~A service marked with propagatesTransaction MUST be dispatched under any propagated (client) transaction.
[POL90016]	~~If a service is marked as transactedOneWay, any OneWay invocation message MUST be received from the transport binding in a transacted fashion, under the target service's global transaction.~~Use of the ***propagatesTransaction*** intent on a service implies that the service binding MUST be capable of receiving a transaction context.
[POL90017]	~~If the component is not configured to run under a global transaction or if the binding does not support transactional message receipt, then a service MUST NOT be marked as transactedOneWay.~~A service marked with suspendsTransaction MUST NOT be dispatched under any propagated (client) transaction.
[POL90019]	When applied to a reference indicates that any OneWay invocation messages MUST be sent immediately regardless of any client transaction.~~A service MUST NOT be marked with "propagatesTransaction" if the component is marked with "managedTransaction.local" or with "noManagedTransaction"~~
[POL90020]	When applied to a service indicates that any OneWay invocation

	MUST be received immediately regardless of any target service transaction.~~When a reference is marked with propagatesTransaction, any transaction context under which the client runs MUST be propagated when the reference is used for a request-response interaction~~
[POL90022]	A service marked with propagatesTransaction MUST be dispatched under any propagated (client) transaction.~~When a reference is marked with suspendsTransaction, any transaction context under which the client runs MUST NOT be propagated when the reference is used.~~
[POL90023]	Use of the ***propagatesTransaction*** intent on a service implies that the service binding MUST be capable of receiving a transaction context.~~A reference MUST NOT be marked with propagatesTransaction if component is marked with "ManagedTransaction.local" or with "noManagedTransaction"~~
[POL90024]	~~A service marked with suspendsTransaction~~Transaction context MUST NOT be ~~dispatched under any~~ propagated ~~(client) transaction.~~on OneWay messages.
[POL90025]	~~A service MUST NOT be marked with "propagatesTransaction" if the component is marked with "managedTransaction.local" or with "noManagedTransaction"~~The SCA runtime MUST ignore the propagatesTransaction intent for OneWay methods.
[POL90026]	~~When a reference is marked with propagatesTransaction, any transaction context under which the client runs MUST be propagated when the reference is used for a request-response interaction~~Any global transaction context that is propagated to the hosting SCA runtime MUST NOT be visible to the target component.
[POL90027]	When a reference is marked with suspendsTransaction, any transaction context under which the client runs MUST NOT be propagated when the reference is used.~~If a transactedOneWay intent is combined with the managedTransaction.local or noManagedTransaction implementation intents for either a reference or a service then an error MUST be raised during deployment.~~
[POL90023]	~~A reference MUST NOT be marked with propagatesTransaction if component is marked with "ManagedTransaction.local" or with "noManagedTransaction"~~When *SOAP* is present, an SCA Runtime MUST use the SOAP messaging model to deliver messages.
[POL90024]	Transaction context MUST NOT be propagated on OneWay messages.~~When a *SOAP* intent is qualified with *1_1* or *1_2*, then SOAP version 1.2 or SOAP version 1.2 respectively MUST be used to deliver messages.~~
[POL90025]	The SCA runtime MUST ignore the propagatesTransaction intent for OneWay methods.~~When *JMS* is present, an SCA Runtime MUST ensure that the binding used to send and receive messages supports the JMS API.~~
[POL90027]	~~If a transactedOneWay intent is combined with the managedTransaction.local or noManagedTransaction implementation intents for either a reference or a service then an error MUST be raised during deployment.~~The *noListener* intent

	MUST only be declared on a @requires attribute of a reference.
[POL90028]	The **transactedOneWay** intent MUST NOT be attached to a request/response operation.~~When *noListener* is present, an SCA Runtime MUST not establish any connection from a service to a client.~~
[POL90029]	The **immediateOneWay** intent MUST NOT be attached to a request/response operation.~~An SCA runtime MUST reject a composite file that does not conform to the sca-policy-1.1.xsd schema.~~
[POL90030]	The ***asyncInvocation*** intent and the ***propagatesTransaction*** intent MUST NOT be applied to the same service or reference operation.
[POL90031]	When the ***asyncInvocation*** intent is applied to an SCA service, the SCA runtime MUST behave as if the ***suspendsTransaction*** intent is also applied to the service.
[POL100001]	When *SOAP* is present, an SCA Runtime MUST use the SOAP messaging model to deliver messages.
[POL100002]	When a *SOAP* intent is qualified with *1_1* or *1_2*, then SOAP version 1.1 or SOAP version 1.2 respectively MUST be used to deliver messages.
[POL100003]	When *JMS* is present, an SCA Runtime MUST ensure that the binding used to send and receive messages supports the JMS API.
[POL100004]	The *noListener* intent MUST only be declared on a @requires attribute of a reference.
[POL100005]	When *noListener* is present, an SCA Runtime MUST not establish any connection from a service to a client.
[POL100006]	When *EJB* is present, an SCA Runtime MUST ensure that the binding used to send and receive messages supports the EJB API.
[POL100007]	The SCA Runtime MUST ignore the asyncInvocation intent for one way operations.
[POL110001]	An SCA runtime MUST reject a composite file that does not conform to the sca-policy-1.1.xsd schema.

Table 0-1: SCA Policy Normative Statements

~~D.~~ Acknowledgements

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

Participant Name	Affiliation
Jeff Anderson	Deloitte Consulting LLP
Ron Barack	SAP AG*
Michael Beisiegel	IBM
Vladislav Bezrukov	SAP AG*
Henning Blohm	SAP AG*
David Booz	IBM
Fred Carter	AmberPoint
Tai-Hsing Cha	TIBCO Software Inc.
Martin Chapman	Oracle Corporation
Mike Edwards	IBM
Raymond Feng	IBM
Billy Feng	Primeton Technologies, Inc.
Robert Freund	Hitachi, Ltd.
Murty Gurajada	TIBCO Software Inc.
Simon Holdsworth	IBM
Michael Kanaley	TIBCO Software Inc.
Anish Karmarkar	Oracle Corporation
Nickolas~~Nickolaos~~ Kavantzas	Oracle Corporation
Rainer Kerth	SAP AG*
Pundalik Kudapkar	TIBCO Software Inc.
Meeraj Kunnumpurath	Individual
Rich Levinson	Oracle Corporation
Mark Little	Red Hat
Ashok Malhotra	Oracle Corporation
Jim Marino	Individual
Jeff Mischkinsky	Oracle Corporation
Dale Moberg	Axway Software*
Simon Nash	Individual
Bob Natale	Mitre Corporation*
Eisaku Nishiyama	Hitachi, Ltd.
Sanjay Patil	SAP AG*
Plamen Pavlov	SAP AG*
Martin Raepple	SAP AG*
Fabian Ritzmann	Sun Microsystems
Ian Robinson	IBM
Scott Vorthmann	TIBCO Software Inc.
Eric Wells	Hitachi, Ltd.
Prasad Yendluri	Software AG, Inc.*
Alexander Zubev	SAP AG*

~~E.~~ Revision History

[optional; should not be included in OASIS Standards]

Revision	Date	Editor	Changes Made
2	Nov 2, 2007	David Booz	Inclusion of OSOA errata and Issue 8
3	Nov 5, 2007	David Booz	Applied resolution of Issue 7, to Section 4.1 and 4.10. Fixed misc. typos/grammatical items.
4	Mar 10, 2008	David Booz	Inclusion of OSOA Transaction specification as Chapter 11. There are no textual changes other than formatting.
5	Apr 28 2008	Ashok Malhotra	Added resolutions to issues 17, 18, 24, 29, 37, 39 and 40,
6	July 7 2008	Mike Edwards	Added resolution for Issue 38
7	Aug 15 2008	David Booz	Applied Issue 26, 27
8	Sept 8 2008	Mike Edwards	Applied resolution for Issue 15
9	Oct 17 2008	David Booz	Various formatting changes Applied 22 – Deleted text in Ch 9 Applied 42 – In section 3.3 Applied 46 – Many sections Applied 52,55 – Many sections Applied 53 – In section 3.3 Applied 56 – In section 3.1 Applied 58 – Many sections
10	Nov 26	David Booz	Applied camelCase words from Liason Applied 54 – many sections Applied 59 – section 4.2, 4.4.2 Applied 60 – section 8.1 Applied 61 – section 4.10, 4.12 Applied 63 – section 9
11	Dec 10	Mike Edwards	Applied 44 - section 3.1, 3.2 (new), 5.0, A.1 Renamed file to sca-policy-1.1-spec-CD01-Rev11
12	Dec 25	Ashok Malhotra	Added RFC 2119 keywords Renamed file to sca-policy-1.1-spec-CD01-Rev12
13	Feb 06 2009	Mike Edwards, Eric	All changes accepted

		Wells, Dave Booz	Revision of the RFC 2119 keywords and the set of normative statements - done in drafts a through g
14	Feb 10 2009	Mike Edwards	All changes accepted, comments removed.
15	Feb 10 2009	Mike Edwards	Issue 64 - Sections A1, B, 10, 9, 8
16	Feb 12, 2009	Ashok Malhotra	Issue 5 The single sca namespace is listed on the title page. Issue 32 clientAuthentication and serverAuthentication Issue 35 Conformance targets added to Appendix C Issue 48 Transaction defaults are not optional Issue 66 Tighten schema for intent Issue 67 Remove 'conversational'
17	Feb 16, 2009	Dave Booz	Issues 57, 69, 70, 71
CD02	Feb 21, 2009	Dave Booz	Editorial changes to make a CD
CD02-rev1	April 7, 2009	Dave Booz	Applied 72, 74,75,77
CD02-rev2	July 21, 2009	Dave Booz	Applied 81,84,85,86,95,96,98,99
CD02-rev3	Aug 12, 2009	Dave Booz	Applied 73,76,78,80,82,83,88,102
CD03-rev4	Sept 3, 2009	Dave Booz	Editorial cleanup to match OASIS templates
CD02-rev5	Nov 9, 2009	Dave Booz	Fixed latest URLs Applied: 79, 87, 90, 97, 100, 101, 103, 106, 107, 108
CD02-rev6	Nov 17, 2009	Dave Booz	Applied 94, 109
CD02-rev7	Jan 1, 2010	Dave Booz	Updated namespace to latest assembly Applied issues: 79,110,111,112,113,114,115
CD02-rev8	Mar 17, 2010	Dave Booz	Applied issue 93 Editorial updates to prepare for next CD
CD02-rev9	April 8, 2010	Ashok Malhotra, Dave Booz	More Editorial cleanup
CD03	May 5, 2010	Dave Booz	Applied 117, Front Matter and TOC updates