THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL

Web Service Contract Design & Versioning for SOA

Thomas Erl, Anish Karmarkar, Priscilla Walmsley, Hugo Haas, Umit Yalcinalp, Canyang (Kevin) Liu, David Orchard, Andre Tost, James Pasley

PRENTICE HALL

Erl_00FM.qxd 8/9/08 11:14 AM Page v

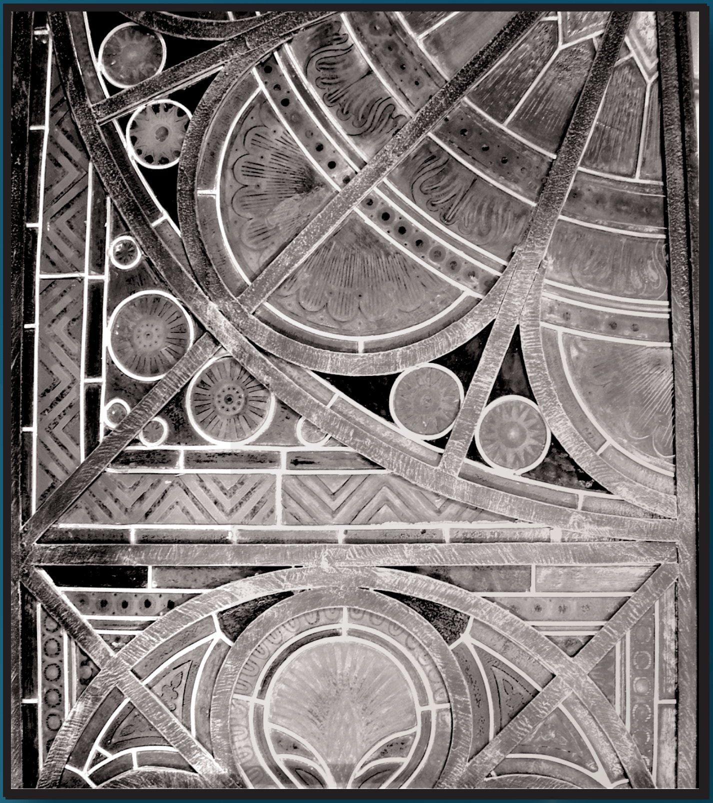
Web Service Contract Design and Versioning for SOA

Thomas Erl, Anish Karmarkar, Priscilla Walmsley,
Hugo Haas, Umit Yalcinalp, Canyang Kevin Liu,
David Orchard, Andre Tost, James Pasley

PRENTICE HALL

PRENTICE HALL
UPPER SADDLE RIVER, NJ • BOSTON • INDIANAPOLIS • SAN FRANCISCO
NEW YORK • TORONTO • MONTREAL • LONDON • MUNICH • PARIS • MADRID
CAPETOWN • SYDNEY • TOKYO • SINGAPORE • MEXICO CITY

Erl_Pt2.qxd 7/30/08 3:08 PM Page 1



Part II

Advanced Service Contract Design

Chapter 12: Advanced XML Schema Part I: Message Flexibility, and Type Inheritance and Composition

Chapter 13: Advanced XML Schema Part II: Reusability, Derived Types, and Relational Design

Chapter 14: Advanced WSDL Part I: Modularization, Extensibility, MEPs, and Asynchrony

Chapter 15: Advanced WSDL Part II: Message Dispatch, Service Instance Identification, and Non-SOAP HTTP Binding

Chapter 16: Advanced WS-Policy Part I: Policy Centralization and Nested, Parameterized, and Ignorable Assertions

Chapter 17: Advanced WS-Policy Part II: Custom Policy Assertion Design, Runtime Representation, and Compatibility

Chapter 18: Advanced Message Design Part I: WS-Addressing Vocabularies

Chapter 19: Advanced Message Design Part II: WS-Addressing Rules and Design Techniques

Erl_16.qxd 8/7/08 9:55 AM Page 1

Chapter 16

Advanced WS-Policy Part I: Policy Centralization and Nested, Parameterized, and Ignorable Assertions

16.1 Reusability and Policy Centralization

16.2 Nested and Parameterized Assertions

16.3 Ignorable Assertions

16.4 Concurrent Policy-Enabled Contracts

Though a simple and small language, WS-Policy allows for complex representations of policies. This chapter explores new design-time options for building sophisticated policy expression structures and entire policy definition architectures, and also discusses the implications of these designs by runtime processors.

16.1 Reusability and Policy Centralization

We concluded Chapter 10 with a look at policies that are embedded within a WSDL definition and also attached to policy subjects. Those embedded policy expressions can be reused within the scope of a WSDL definition by having multiple `wsp:PolicyReference` elements reference the same `wsp:Policy` construct via its `name`, `wsu:Id` or `xml:Id` attribute.

We can consider this a form of intra-document reuse, where the scope of reusability is limited to one WSDL definition. In this section, we will be exploring the physical centralization of policy expressions into separate policy documents that we'll refer to as *policy definitions*.

NOTE

The following sections focus on the "controlled reuse" of policies within pre-defined service inventory boundaries. While this is a common approach to sharing policies across WSDL definitions, it does not preclude you from simply reusing policies to whatever extent you choose, regardless of centralization considerations.

Policy Centralization and Policy Definitions

As you might recall, you can establish many-to-many relationships between WSDL definitions and XML schemas where one XML schema is reused (imported or included) within multiple WSDL definitions, and/or a single WSDL definition can reference and pull in content from multiple XML schemas. This flexibility allows you to establish a Web service contract architecture that can leverage and foster reuse.

Similar reuse opportunities are available when exploring relationships between WSDL definitions and policy definitions. In fact, grouping common policy expressions within separate policy definition documents is the basis of an SOA design pattern called Policy Centralization, which we explain shortly.

In Chapter 14 we introduced the notion of creating reusable schemas that contained common types. Some of those types were limited to a domain (like the purchasing-specific types in the Purchasing Common schema), while others were useful on a global basis (such as with the generic types in the Common schema).

When it comes to creating physically separate policy definition documents, we have the same options, as follows:

Domain Policy Definition

You can create policy expressions that apply to a subset of the services within a given service inventory. These policies are still reusable because they are applicable to multiple Web service contracts, but they are limited in scope to a particular domain. As with creating domain-specific common schema types, domain-level policies are often also related to a business domain. However, it is up to you to create your own domains based on whatever requirements you have. For example, you could establish a policy domain specific to a transport protocol.

NOTE

In the upcoming case study example, we will be creating a domain policy definition that contains policy expressions for the purchasing domain only.

Global Policy Definitions

Global policies are expected to apply to all services within a particular governance boundary. Therefore, global policy definitions often contain very broad and generalized policy expressions.

The scope implied by the use of the term “global” depends on the scope of the underlying service inventory. As explained early on in Chapter 3, you can establish pools of services (service inventories) that are independently standardized and governed. Sometimes, an inventory is enterprise-wide, whereas other times it only represents a domain within the enterprise.

In the latter case, a global policy definition would be applicable to just the scope of the domain inventory. A domain policy definition would then apply to a subset of the service contracts within the domain inventory. So again, when we call a policy definition “global,” we mean that it is global within one service inventory boundary only.

NOTE

You can also look into creating policies that span multiple domain service inventories. This may or may not be a good idea, depending on how complex the architecture becomes and also on the governance impact this may result in. However, if this is something you do decide to explore, you can further qualify these types of polices as "master global policies" or just "master policies."

Policy Centralization

This pattern (which is similar in concept to the Schema Centralization pattern) simply advocates creating the domain and global policy definitions we just described. It results in a system whereby policies can be consistently enforced across several Web service contracts. This reduces redundant policy content within a service inventory and further allows policies to be centrally maintained.

An added benefit is that the policy documents can more easily be owned by custodians that do not necessarily have to be involved with the governance of the WSDL definitions. This type of freedom is conducive to evolving technical policies in tandem with actual business policies and also provides policy custodians the option to use their own, preferred tools.

NOTE

The techniques in this section can also be applied to WSDL definition-specific policies. Instead of embedding non-reusable policy expressions within WSDL documents, you can also isolate them in separate policy definition documents to allow them to be maintained in a physically separate file.

Designing External WS-Policy Definitions

Several approaches exist for establishing a relationship between a WSDL definition document and a separate WS-Policy definition document. However, not all approaches are supported by all vendor platforms, and some platforms may not support external policy reuse at all. Therefore, it is very important that you investigate the environment in which you plan to deploy external policy definitions before deciding on any of the techniques described in the upcoming sections.

Policy Processors

Unlike XML and WSDL processors that are widely established, runtime programs that perform WS-Policy-specific processing have not been formally defined and their behavior can therefore vary across vendor platforms. In some cases, it may be a dedicated event-driven agent that carries policy processing out, whereas other times this logic may exist as an extension of the overall Web services toolkit and runtime platform. It's therefore helpful to keep this in mind whenever you see the term "policy processing" used in this book.

Using the `wsp:PolicyAttachment` *Element*

A common means of sharing policy expression code is to place into its own WS-Policy definition via the `wsp:PolicyAttachment` element, as shown here:

```
<wsp:PolicyAttachment>
  ...
  <wsp:Policy>
    <wsam:Addressing/>
    <wsrmp:RMAssertion optional="true"/>
  </wsp:Policy>
</wsp:PolicyAttachment>
```

Example 16.1

A `wsp:PolicyAttachment` construct containing a policy expression. This code presumably resides in a separate policy definition document.

Here we can see that the `wsp:PolicyAttachment` construct simply wraps around an existing policy expression. The ellipsis at the top indicates room for any of the following additional child elements:

- `wsp:AppliesTo` – This element is used to indicate what part of the WSDL document the policy expression applies to.
- `wsp:Policy` or `wsp:PolicyReference` – These elements have been previously explained. As child elements to `wsp:PolicyAttachment`, they designate the specific policy expression that will be applied to the policy subject identified in the `wsp:AppliesTo` element.

The content of a `wsp:AppliesTo` element can be any element. This makes the external referencing mechanism quite powerful in that you can technically target any part of a

WSDL definition. This element uses a further `wsp:URI` element that allows for the identification of the target element via a URL statement.

In the following example we've populated the `wsp:PolicyAttachment` construct with `wsp:AppliesTo` and `wsp:URI` child elements that indicate that the policy expression at the bottom of the `wsp:PolicyAttachment` construct will be applied to the `endpoint` element of the WSDL 2.0 binding for the Purchase Order service.

```
<wsp:PolicyAttachment>
  <wsp:AppliesTo>
    <wsp:URI>
      http://actioncon.com/purchaseOrder.wsdl20
      #wsdl.endpoint(PurchaseOrderService/Endpoint)
    </wsp:URI>
  </wsp:AppliesTo>
  <wsp:Policy>
    <wsam:Addressing/>
    <wsrmp:RMAssertion optional="true"/>
  </wsp:Policy>
</wsp:PolicyAttachment>
```

Example 16.2
The `wsp:AppliesTo` and `wsp:URI` child elements with a URL that identifies the target element of the policy.

NOTE

A key architectural consideration with this approach is that it places control of what WSDL definitions and subjects the policy applies to in the hands of the policy definition owner. This differs from upcoming alternatives where policies exist in separate documents but their application is determined within WSDL definitions instead.

Using the `wsp:PolicyURIs` *Attribute*

An alternative attachment method documented in the WS-Policy specification is the use of the `wsp:PolicyURIs` attribute which can simply be added (as an extensibility attribute) to any valid policy attachment point within a WSDL definition, as shown here:

```
<portType name="ptPurchaseOrder"
  wsp:PolicyURIs="custom:commonPolicies.xml">
  <operation name="opSubmitOrder">
    <input message="tns:msgSubmitOrderRequest"/>
    <output message="tns:msgsubmitOrderResponse"/>
  </operation>
  ...
</portType>
```

Example 16.3
The `portType` element is extended with the `wsp:PolicyURIs` attribute.

In this example, the `ptPurchaseOrder portType` element is attached to the policies residing in the commonPolicies.xml document. Note that the `wsp:PolicyURIs` attribute can contain a list of URI addresses that point to multiple physically separate policy definition documents.

Using the `Name` *and* `URI` *Attributes*

The WS-Policy language provides a `Name` attribute that allows you to assign a separate ID value to a `wsp:Policy` element specifically for external references. Even though this attribute is called "Name," the value needs to be an IRI.

A `wsp:PolicyReference` element located in a separate document (most likely a WSDL definition) can then point to the external policy expression via a `URI` attribute that is populated with the same value as the `wsp:Policy` element's `Name` attribute.

So let's imagine that the following policy expression is located in a document called commonPolicies.xml:

```
<wsp:Policy Name="http://actioncon.com/policies/common">
  <wsam:Addressing/>
</wsp:Policy>
```

Example 16.4
A `wsp:Policy` construct residing within a separate policy definition document. The `Name` attribute provides the policy expression with an externally "referenceable" identifier.

The value of the `Name` attribute gives this `wsp:Policy` construct a reference ID that can be used by `wsp:PolicyReference` elements to point to and reuse the policy expression.

For example, the following `wsp:PolicyReference` statement may reside in a WSDL definition document:

```
<wsp:PolicyReference
  URI="http://actioncon.com/policies/common"/>
```

Example 16.5

A `wsp:PolicyReference` element that points to the previously displayed wsp:Policy construct via the `URI` attribute.

NOTE

The mechanics behind performing the actual inclusion of the policy expression into the WSDL definition is up to your runtime and service hosting platform. If you intend to use this approach, be sure to confirm that your policy processors support these attributes. You may also want to consider using (or you may be required to use) XPointer to enable cross-document inclusion.

Wrapping Policies Within WSDL Definitions

An alternative means of creating common policy definition documents is to place the reusable policy expressions into a separate WSDL definition document. This is by no means an "official" approach documented in the WS-Policy specifications, but should instead be considered a possible workaround if support for the preceding techniques is not provided by your platform.

With this approach, you may be able to simply use the existing WSDL `import` or `include` elements to pull in the contents of the external WSDL document containing the policy expressions.

In this case, you would place the policy expression within a WSDL `definitions` construct as follows:

```
<definitions targetNamespace=
  "http://actioncon.com/policies/common">
  <wsp:Policy wsu:Id="addressing-policy">
    <wsam:Addressing/>
  </wsp:Policy>
  ...
</definitions>
```

Example 16.6

A WSDL `definitions` construct that acts as a container for common policies.

As shown here, you can optionally assign the WSDL definition its own target namespace value.

NOTE

In this case we used the value assigned to the `wsp:Policy` element's `Name` attribute from the previous example as the target namespace value. However, this is just incidental. You can create whatever namespace value you like.

Within the WSDL definition that needs to pull in this policy, you can then add a standard WSDL `import` element that points to the WSDL definition acting as the policy definition, as follows:

```
<definitions targetNamespace=
  "http://actioncon.com/contract/PurchaseOrder" ...>
  <import
    namespace="http://actioncon.com/policies/common"
    location="http://actioncon.com/policies/common"/>
  ...
  <wsp:PolicyReference URI="#addressing-policy"/>
  ...
</definitions>
```

Example 16.7

An example of a WSDL definition that includes a `wsp:PolicyReference` statement that points to an imported `wsp:Policy` construct. If you check back to the *Attaching Policies to WSDL Definitions* section from Chapter 10, you can see that this code resembles the same syntax used for local references within a WSDL document.

Note how the `wsp:Policy` element from Example 16.6 does not use a `Name` attribute. This is because it is not expecting to be externally referenced. Via the WSDL import mechanism, the policy expression is brought into the WSDL definition and then referenced as though it was a local part of the document. This is also why the `URI` attribute of the `wsp:PolicyReference` element contains only a local pointer (based on the convention of using the hash mark: "`#`").

NOTE

One potential problem with this approach is the enforcement of the "uniqueness" of the `wsp:Policy` element's `xml:Id` or `wsu:Id` attribute value across multiple WSDL documents. If this issue cannot be resolved by your platform, the use of XML Include or XPointer may need to be considered.

CASE STUDY EXAMPLE

Soon after refining his Web service contracts with some of the more advanced XML Schema and WSDL features, Steve is called into a meeting with the Kevin, the CEO, and Donna, the president of ActionCon.

This meeting concerns Steve. It's unusual to have a formal meeting with Kevin, and he hasn't even met Donna yet. His first thought is that his department is being downsized and Steve begins to think about how he should update his resume as he heads toward the meeting room.

However, his fears soon disappear when he hears the news. Apparently, there have been some recent security breaches resulting in stolen corporate financial data and one attempted malicious attack on the ActionCon data warehouse that was luckily caught and countered in time.

As a result of these events, Steve is told that all Web services that handle financial data must be fully secured and must also require that outside consumers comply with the use of certain security options. Steve is informed that an external security consulting company has been hired to perform an audit and that they will soon be providing him with a list of requirements that will impact the design of his Web services.

Subsequent to the meeting, Steve begins to re-investigate the use of the WS-Policy framework. In his previous work with policies (from Chapter 10), Steve successfully attached a policy to one of his Web service contracts. But given the scope of the upcoming security requirements, he now turns his attention to establishing a centralized policy architecture, the initial draft of which is displayed in Figure 16.1.

He knows that there will be the need for security policies that affect Web services that process financial information. As shown on the right side of Figure 16.1, he establishes a logical finance domain that will be supported by a domain policy definition comprised of policy expressions that will apply to all Web services that handle financial data. Currently, both his Purchase Order and Invoice services would fall within this domain.

The following example shows a basic skeleton outline of the policy expression that will reside in the financePolicies.xml document:

```
<wsp:Policy Name="http://actioncon.com/policies/finance">
  <!— finance-related policy assertions —>
</wsp:Policy>
```

Example 16.8
The start of a domain policy definition containing finance-related assertions.

Erl_16.qxd 8/7/08 9:55 AM Page 11

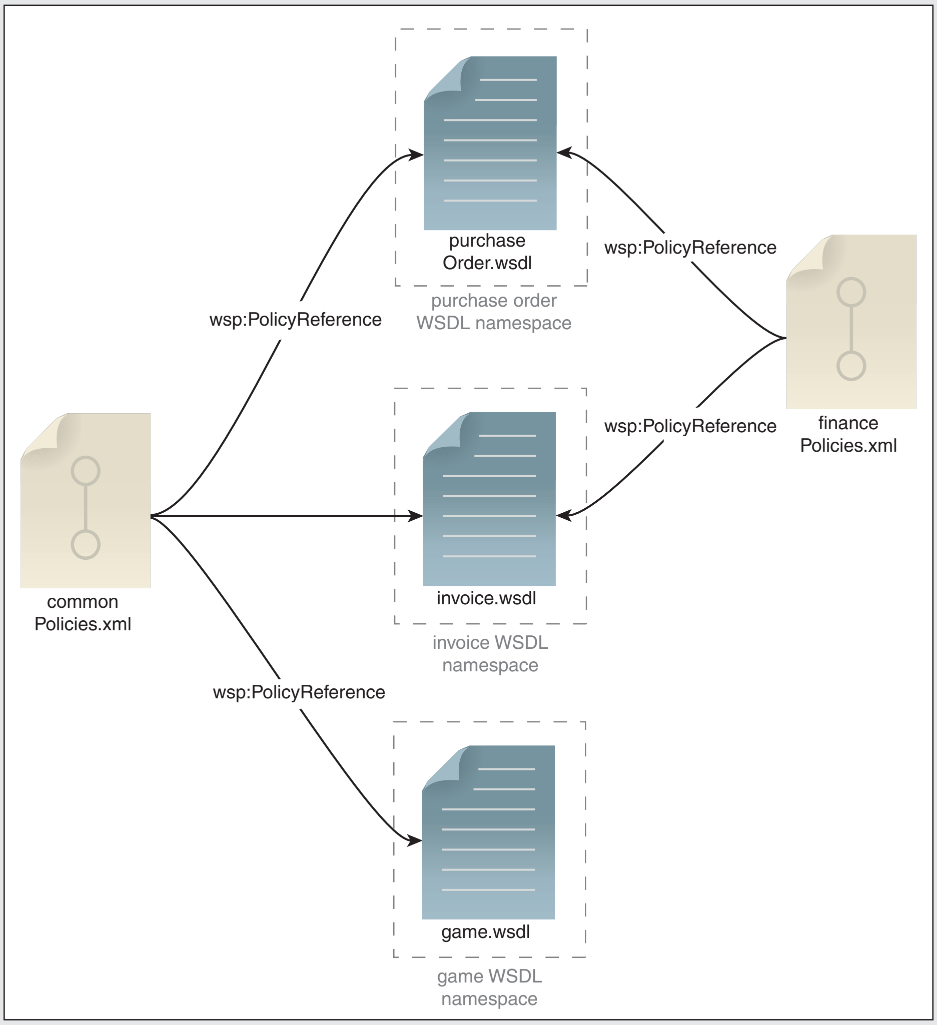


Figure 16.1

A domain policy definition (right) associated with finance-related Web services, and a global policy definition (left) providing common policies that apply to all services.

NOTE

The contents of the financePolicies.xml domain policy definition are developed in the upcoming *Case Study Example: Nested and Parameterized Assertions* section.

As indicated by the left side of his policy centralization architecture, Steve decides that he might as well also establish a global policy definition that will provide general policy expressions that apply to all Web services within his planned inventory.

Based on a recent enterprise design standard handed down by the CTO, Steve already knows that one global policy will be to require that all Web service contracts contain an assertion that requires consumer programs to support WS-Addressing headers. Therefore, his initial policy expression for the commonPolicies.xml global policy definition looks like this:

```
<wsp:Policy Name="http://actioncon.com/policies/common">
  <wsam:Addressing/>
</wsp:Policy>
```

Example 16.9
By virtue of the fact that this simple policy expression is part of a global policy definition, it is expected to extend all Web service contracts within Steve's planned service inventory.

Common Policy Centralization Challenges

In order to facilitate such a centralized policy architecture, a number of considerations need to be taken into account, several of which might impose significant challenges upon an IT department:

- New governance processes are required to ensure that global and domain policy definitions are maintained and to further guarantee that changes to these policies will not negatively impact any of the Web service contracts they apply to.
- Standard processes are required for project teams to reliably obtain domain and global WS-Policy definition documents. This means that a discovery process ordinarily geared toward WSDL documents will now need to extend to WS-Policy definitions.
- Conflicts may exist between overlapping policies, such as when a global policy introduces a constraint that is contrary to a constraint provided by a domain-level policy. These conflicts need to be taken into account whenever new domain or global polices are added or modified.
- One centralization challenge related in particular to the use of the WS-PolicyAttachment mechanism is that the infrastructure needs to know where to get the

policies from. Often, a middleware platform, such as an ESB, is required (as explained shortly).

Finally, as mentioned in the previous section, not all service runtime platforms support external referencing to WS-Policy definitions that exist as standalone XML documents. Without robust support for sharing policies, it is difficult to achieve meaningful centralization.

NOTE

An additional design-related challenge to achieving a centralized policy architecture is knowing in advance where to attach policies to. Because policies can be associated with different attachment points within different policy subjects, it's impossible to know ahead of time what a domain or global policy will apply to. When manually maintaining policy-enabled WSDL definitions in a centralized architecture, one approach is to "pre-attach" a `wsp:PolicyReference` element to some or all potential attachment points in a WSDL `definitions` construct and for most of these elements to initially point to empty policy expressions (`wsp:Policy` constructs with no assertions).

This establishes a relationship between the WSDL document and a series of centralized policy definition documents and allows policy custodians to add new expressions without having to revisit the WSDL definitions. However, this is not a common or recommended practice. It can result in an awkward architecture, and in some environments, it may add runtime processing cycles as the platform hosting the Web services may try to resolve and check for external policies that aren't there each time the service is invoked. Furthermore, conflicts can arise when empty and non-empty policies are combined across policy attachment points.

Policy Centralization and ESBs

Several platforms (especially those provided by modern Enterprise Service Bus products) are equipped with built-in support for policy centralization. You simply fill out a form to associate a centralized policy expression with your services, and the platform takes care of the rest. This can be an extremely convenient and powerful means of achieving an effective centralized policy architecture; however, it often comes with the trade-off that your Web service contracts must form dependencies on proprietary features that may be difficult to move away from if you ever want to change or diversify your service inventory architecture.

SUMMARY OF KEY POINTS

- Policy expressions can be isolated into policy definition documents that can be shared and reused across WSDL documents, just like XML Schema definitions.
- One approach involves using the `wsp:Policy` element's `Name` attribute to establish an externally referenceable identifier. However, you need to ensure that the service inventory architecture supports this type of external references.
- Another approach is to create policy definition documents as WSDL definitions that are then imported into other WSDL definitions. This technique can be used when there is insufficient product support for regular external references.

16.2 Nested and Parameterized Assertions

In Chapter 10 we covered simple assertions, operators, and expressions. In this section, we'll take a look at how these parts can be further combined and extended. Specifically, we'll explore how policy expressions can be nested within each other and how they can be designed to accept and respond to parameter data provided by the consumer.

Nested Policy Assertions

Policy assertions can be structured around parent-child relationships whereby a child policy assertion is nested within a parent assertion. As shown in the following example, assertion `ex:Assertion2` is nested within `ex:Assertion1`:

```
<wsp:Policy>
  <wsp:ExactlyOne>
    <wsp:All>
      <ex:Assertion1>
        <wsp:Policy>
          <wsp:ExactlyOne>
            <wsp:All>
              <ex:Assertion2/>
            </wsp:All>
          </wsp:ExactlyOne>
        </wsp:Policy>
      </ex:Assertion1>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
```

Example 16.10
One assertion nested within another.

So, why would you want to nest a policy assertion? There's a simple, two-part answer.

An assertion needs to be nested when:

- the behavior of the nested assertion is dependent on the parent assertion (or vice versa), and/or
- the parent and child assertions apply to the same targets (attachment points)

Sometimes a policy assertion will be defined to *extend* an existing assertion. In this case, it will likely need to be nested within the existing assertion because it is directly dependent on the assertion it is extending. Furthermore, for this type of structure to make sense, both parent and child assertions need to be targeting the same policy subject within the WSDL definition.

As you may have noticed in the previous sample code, the syntax requirement for nesting assertions is that the child assertion be wrapped in its own `wsp:Policy` construct. You may be wondering why this is actually required.

For example, why can't you just do this:

```
<wsp:Policy>
  <wsp:ExactlyOne>
    <wsp:All>
      <ex:Assertion1>
        <ex:Assertion2/>
      </ex:Assertion1>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Example 16.11
An attempt at nesting an assertion that does not result in an actual, nested assertion.

The reason the approach in this example doesn't result in a nested assertion has to do with how runtime policy processors work. (Specifically, this is related to how these processors perform "policy matching.") The policy processor looks for the wsp:Policy element and when it finds it, it only cares about the first assertion element it encounters.

To the processor, ex:Assertion1 represents the policy assertion and it has no interest in what lies within the ex:Assertion1 construct (unless there happens to be a nested wsp:Policy or wsp:PolicyReference element). Therefore, to communicate to the processor that there are actually two separate policy assertions, we need to add two separate wsp:Policy constructs (as we did in Example 16.10).

Parameterized Assertions

Even though we just dismissed Example 16.11 as not representing a nested policy, it does actually demonstrate another important type of policy structure. Because the ex:Assertion2 element is not its own policy, it is considered a parameter of ex:Assertion1. This means that whatever part of the runtime environment will be responsible for processing the ex:Assertion1 policy will receive the value of ex:Assertion2 as input.

There are, of course, other ways to create parameterized assertions. In fact, we previewed one in Chapter 10. You might recall the following example:

```
<wsp:Policy>
  <argt:responseGuarantee>
    <argt:responseInMilliseconds>
      50
    </argt:responseInMilliseconds>
  </argt:responseGuarantee>
</wsp:Policy>
```

Example 16.12
A parameterized argt:responseGuarantee policy assertion.

In this case, all of the contents of the argt:responseGuarantee construct are considered parameters. Given that this is a custom policy assertion (not one that originated with an industry standard), there will need to be custom service logic that processes these parameters at runtime. Also, development tools will likely not be able to check for policy compatibility using these custom assertions.

Let's now check out a more detailed case study example with both nested and parameterized policy assertions based on industry standards.

NOTE

Unlike nested assertions for which support is quite common, not all platforms provide support for parameterized assertions.

CASE STUDY EXAMPLE

Nested and Parameterized WS-SecurityPolicy Assertions

NOTE

The following case study example makes references to the WS-Security and WS-SecurityPolicy languages, which are not explained in this book. You do not need to be familiar with these standards to understand this example. You can ignore the security-related terminology and just focus on the highlighted assertions that demonstrate nested and parameterized structures. (If you are interested in learning more about this standard, be sure to visit www.soaspecs.com.)

Note also that in this example we will use the `sp:` prefix to represent assertions that are defined in the WS-SecurityPolicy specification.

Earlier (in Chapter 14) Steve was told that there were insufficient funds to establish the infrastructure to support a WS-Security framework. Due to the recent security breaches, establishing this framework has now become a priority.

Specifically, the security audit carried out by the external consultants produces the following security requirements that affect all Web services that process financial data:

- Consumers must now access all Web services using transport or message-level security.
- For transport-level security, Web services need to be accessed via SSL with transport-level tokens.
- For message-level security, Web services need to be accessed using X509 tokens for authentication and all messages must be signed and encrypted.

In response to these requirements, Steve decides to design a policy alternative for his newly created domain policy definition that gives consumers a choice between using either transport- or message-level security. He uses two pre-defined assertions from the WS-SecurityPolicy language to express these options, as follows:

```
<wsp:Policy Name="http://actioncon.com/policies/finance">
  <wsp:ExactlyOne>
    <sp:TransportBinding/>
    <sp:SymmetricBinding/>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Example 16.13
The policy expression from the domain policy definition now comprised of a policy alternative offering a choice between two security-related assertions.

Steve looks at his newly created policy expression and thinks to himself, "It can't be that easy…" He begins reading the WS-SecurityPolicy specification and soon finds out that it isn't.

He discovers that for both assertions additional properties need to be defined. For example, ActionCon requires a specific transport token (https) for transport level security which relies on timestamps to be present in the security header of a SOAP envelope. Additionally, a specific algorithm suite (Basic264) for cryptography needs to be supported.

All of these required properties are defined as three additional nested assertions as per the highlighted parts of this example:

```
<wsp:Policy Name="http://actioncon.com/policies/finance">
  <wsp:ExactlyOne>
    <sp:TransportBinding>
      <wsp:Policy>
        <sp:TransportToken>
          <wsp:Policy>
            <sp:HttpsToken>
              ...tokens...
            </sp:HttpsToken>
          </wsp:Policy>
        </sp:TransportToken>
      <sp:IncludeTimestamp/>
      <sp:AlgorithmSuite>
        <wsp:Policy>
          <sp:Basic256/>
        </wsp:Policy>
      </sp:AlgoritmSuite>
    </wsp:Policy>
    </sp:TransportBinding>
    ...
  </wsp:Policy>
```

Example 16.14
The sp:TransportBinding policy assertion containing three nested policy assertions.

Steve now turns his attention to the sp:SymmetricBinding assertion that represents the message-level security alternative. This assertion actually contains *seven* nested assertions within three nested wsp:Policy layers, as follows:

```
<wsp:Policy Name="http://actioncon.com/policies/finance">
  <wsp:ExactlyOne>
    ...
    <sp:SymmetricBinding>
      <wsp:Policy>
        <sp:ProtectionToken>
          <wsp:Policy>
            <sp:X509Token sp:IncludeToken=
              "http://schemas.xmlsoap.org/ws/2005
              /07/securitypolicy/IncludeToken/Never">
              <wsp:Policy>
                ...details of X509 token...
              </wsp:Policy>
            </sp:X509Token>
          </wsp:Policy>
        </sp:ProtectionToken>
        <sp:AlgorithmSuite>
          <wsp:Policy>
            <sp:Basic256/>
          </wsp:Policy>
        </sp:AlgorithmSuite>
        <sp:Layout>
          ...layout details...
        </sp:Layout>
        <sp:IncludeTimestamp/>
        <sp:OnlySignEntireHeadersAndBody/>
      </wsp:Policy>
    </sp:SymmetricBinding>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Example 16.15
The sp:SymmetricBinding policy assertion containing seven nested policy assertions.

As a result of this exercise, all of the nested assertions required to fully provide the two security-related policy alternatives are established. But what about parameterized assertions? Some of the nested assertions shown in the previous example are, in fact, also parameterized.

This next example displays the entire policy expression with both policy alternatives. The highlighted parts represent parameters for parameterized assertions:

```
<wsp:Policy Name="http://actioncon.com/policies/finance">
  <wsp:ExactlyOne>
    <sp:TransportBinding>
      <wsp:Policy>
        <sp:TransportToken/>
          <wsp:Policy>
            <sp:HttpsToken>
              ...tokens...
            </sp:HttpsToken>
          </wsp:Policy>
        </sp:TransportToken>
        <sp:IncludeTimestamp/>
        <sp:AlgorithmSuite>
          <wsp:Policy>
            <sp:Basic256/>
          </wsp:Policy>
        </sp:AlgoritmSuite>
      </wsp:Policy>
    </sp:TransportBinding>
    <sp:SymmetricBinding>
      <wsp:Policy>
        <sp:ProtectionToken>
          <wsp:Policy>
            <sp:X509Token sp:IncludeToken=
              "http://schemas.xmlsoap.org/ws/2005
              /07/securitypolicy/IncludeToken/Never">
              <wsp:Policy>
                ...details of X509 token...
              </wsp:Policy>
            </sp:X509Token>
          </wsp:Policy>
        </sp:ProtectionToken>
        <sp:AlgorithmSuite>
          <wsp:Policy>
            <sp:Basic256/>
          </wsp:Policy>
        </sp:AlgorithmSuite>
          <sp:Layout>
            ...layout details...
           </sp:Layout>
          <sp:IncludeTimestamp/>
          <sp:OnlySignEntireHeadersAndBody/>
        </wsp:Policy>
      </sp:SymmetricBinding>
    </sp:ExactlyOne>
  </wsp:Policy>
```

Example 16.16
A policy expression comprised of two security-related policy alternatives.

The italicized text shows where some of the parameter data would be placed, but for simplicity's sake it's been omitted. Note how the `sp:X509Token` element incorporates assertion parameters via attributes. How parameter data is defined and represented within a given assertion depends solely on the design of the underlying policy assertion type.

SUMMARY OF KEY POINTS

- Policy expressions can be nested within each other, thereby allowing for the creation of complex constructs with multiple policy assertions.
- When policy assertion elements are nested or contain data values determined at runtime, the policy is considered to be parameterized.

16.3 Ignorable Assertions

No part of the WS-Policy language has resulted in as much debate as the use of ignorable assertions. In a nutshell, this feature allows you to express a behavior of a Web service as an assertion that consumer programs can simply choose to disregard.

Here's what an ignorable assertion looks like:

```
<wsp:Policy>
  <custom:TraceMessage wsp:Ignorable="true"/>
</wsp:Policy>
```

Example 16.17
A policy expression containing an ignorable policy assertion.

In this little example, `custom:TraceMessage` represents a custom assertion, but the setting of `wsp:Ignorable="true"` makes consumer-side processing of the assertion purely voluntary, meaning that consumers don't need to perform extra processing in order to communicate with the service.

To better understand this attribute, let's begin by comparing it to the `wsp:Optional` attribute we introduced in Chapter 10.

`wsp:Ignorable` vs. `wsp:Optional`

As you might recall, the `wsp:Optional` attribute can also be used to label a policy assertion as not being required. How then is `wsp:Ignorable` different from `wsp:Optional`? The difference between these two attributes has to do with the types of assertions they are applied to.

Applying the `wsp:Optional` *Attribute*

The `wsp:Optional` attribute is generally used as a form of "short hand" to define the equivalent of a policy alternative that offers the consumer a choice of whether to process an assertion or not process anything at all.

Therefore, this attribute tends to communicate to the consumer that:

> "You can choose to comply with the assertion and do the corresponding processing, or you can choose not to and then no assertion-related processing will occur."

A good example of this is when a technology is supported by a Web service, but its use is not mandatory by all consumers, as follows:

```
<wsp:Policy>
  <wsam:Addressing wsp:Optional="true"/>
</wsp:Policy>
```

Example 16.18
A policy assertion with the `wsp:Optional` attribute.

In this case, the Web service indicates that it supports WS-Addressing headers, but consumers don't need to use them if they don't want to.

Applying the `wsp:Ignorable` *Attribute*

What the `wsp:Ignorable` is most often used to communicate about a Web service is some behavior that will be carried out, regardless of whether the consumer program acknowledges or processes it.

In other words, this attribute conveys to the consumer that:

> "By the way, you should be aware of the fact that the Web service will be doing this thing regardless of whether or not you will do anything in response to it."

The `custom:TraceMessage` assertion example we provided at the beginning of this section is appropriate for this attribute. It basically states that incoming messages from the consumer will be traced either way.

Other common applications for this attribute include:

- communicating messaging-related behaviors (stating that messages are being logged or that message details will be retained in memory as state data)
- communicating assurances (such as response time or availability guarantees, or promising non third-party or intermediary involvement)

In fact, you can express anything you like with ignorable assertions. However, this feature does need to be used with caution. See the *Considerations for Using Ignorable Assertions* section for some guidelines.

What About Using `wsp:Optional` *and* `wsp:Ignorable` *Together?*

Here's a brain teaser. The `wsp:Ignorable` attribute is supposed to indicate assertion behavior that will always occur, while the `wsp:Optional` attribute is intended to indicate that an assertion behavior does not have to occur.

So what happens when we create an assertion like this:

```
<wsp:Policy>
  <custom:TraceMessage
    wsp:Ignorable="true"
    wsp:Optional="true"/>
</wsp:Policy>
```

Example 16.19
A policy assertion with both `wsp:Ignorable` and `wsp:Optional` attributes.

A good way to understand the implications of this is to reorganize the assertion into a set of policy alternatives, as follows:

```
<wsp:Policy>
  <wsp:exactlyOne>
    <wsp:All>
      <custom:TraceMessage wsp:Ignorable="true"/>
    </wsp:All>
    <wsp:All/>
  </wsp:exactlyOne>
</wsp:Policy>
```

Example 16.20
The policy expression from Example 16.19 is restructured into a policy alternative.

What we end up with is a policy alternative with no required assertions, which is very similar to just using the `wsp:Optional` attribute on its own. Therefore, this combination is usually considered inappropriate unless the underlying runtime platform provides some proprietary processing that requires the presence of these two attributes or unless there are requirements that the `wsp:Ignorable` attribute be added simply for communication purposes.

NOTE

Another difference between how these two attributes are processed relates to the use of the normal form, which is explained in the upcoming *Normalization* section of this chapter. The `wsp:Optional` attribute is not retained after the normal form is applied (because it is turned into an alternative), whereas the `wsp:Ignorable` attribute remains part of the normal form, as an attribute of the assertions.

Incidentally, when we restructured the policy expression from Example 16.19 into the policy alternative displayed in Example 16.20, we applied normalization.

Using `wsp:Ignorable` to Target Consumers

Because it is the consumer program's responsibility to either ignore or understand and respond to ignorable assertions, you can use this feature to provide policy assertions that are specifically targeted to different types of consumers. This is especially useful when creating custom policy assertions that not all consumers will understand.

For example, you may have an agnostic Web service that gets reused a lot as part of different service compositions. A new composition may require that the service express an assurance that communicates its availability to other services, as follows:

```
<wsp:Policy>
  <custom:Available wsp:Ignorable="true">
    <start>5:00</start>
    <end>23:00</end>
  </custom:Available>
</wsp:Policy>
```

Example 16.21

A parameterized instance of an ignorable policy assertion that states that the Web service is available between 5 AM and 11 PM.

This allows those other services (which act as consumers when they invoke your Web service) to first check the availability assurance assertion prior to attempting invocation.

In this example, the use of the `custom:Available` assertion is only required by that one service composition. All of the other compositions and solutions that reuse the Web service to automate different business processes may not understand or require knowledge of this assertion. And, most importantly, even though the assertion was added well after the Web service has been in production, because it is ignorable, its presence has no effect on any of the existing service consumers (disregarding the fact that consumers who attempt to access this service between 11:00 PM and 5:00 AM will certainly be affected).

CASE STUDY

Adding an Ignorable Domain Policy Expression

Another recommendation that was part of the security audit report (explained in earlier examples) was that all incoming messages containing financial data be logged. These logs will provide a valuable record of service usage and can help trace back any attempted attacks or misuse of a Web service. As an added bonus, the logged data can further be used for diagnostic purposes, fault detection, and will also help provide usage statistics.

When first hearing of this new requirement, Steve boldly asks, “Why can’t we just add logging functionality to the service logic without having to advertise it in the service contract?” It’s a valid point, thinks Steve, especially considering that this functionality is supposed to be added for security purposes.

The security consultants inform Steve that they would like nothing more than to see the logging function be added transparently in the background. However, other ActionCon architects have pointed out that certain service consumer programs sometimes need to send messages with highly sensitive data. In these cases, these consumers require the option of not accessing a Web service that logs incoming messages.

Steve now gets the picture and proceeds to expand his original finance domain policy definition with the ignorable `custom:LogMessage` assertion, as follows:

```
<wsp:Policy Name="http://actioncon.com/policies/finance"
  ...>
  <wsp:All>
    <custom:LogMessage wsp:Ignorable="true"/>
    <wsp:ExactlyOne>
      <sp:TransportBinding>
        ...
      </sp:TransportBinding>
      <sp:SymmetricBinding>
        ...
      </sp:SymmetricBinding>
    </sp:ExactlyOne>
  </wsp:All>
</wsp:Policy>
```

Example 16.22

The ignorable `custom:LogMessage` assertion is added to the finance domain policy definition from the previous case study example.

With this ignorable assertion in place, it is now up to consumer programs to check for its existence to determine whether they want to send a Web service within the finance domain a message or not.

Considerations for Using Ignorable Assertions

You can use the `wsp:Ignorable` attribute as an extension to the Web service contract to communicate pretty much anything you want about a Web service, but that doesn't necessarily always make it a good idea.

As with any part of a Web service contract, you need to respect the Service Loose Coupling design principle to avoid inadvertently allowing implementation details to make their way into the WSDL definition.

Erl_16.qxd 8/7/08 9:55 AM Page 27

For example, an ignorable assertion like this:

```
<wsp:Policy>
  <custom:MyDatabaseIsDB2 wsp:Ignorable="true"/>
</wsp:Policy>
```

Example 16.23
An ignorable policy assertion of questionable value.

…is just not a good idea. Even if there was an immediate requirement to communicate the underlying database product to consumers, using the technical interface is the last place you'd want to do it.

This is an extreme example of a negative type of coupling called Contract-to-Implementation coupling. The problem this leads to is that consumer programs may be developed to bind and process these very implementation-specific assertions, and as soon as you change the implementation, a change to the assertion name will impact the consumers.

Of course, you could parameterize this assertion as follows:

```
<wsp:Policy>
  <custom:Database wsp:Ignorable="true">
    DB2
  </custom:Database>
</wsp:Policy>
```

Example 16.24
A parameterized version of the preceding policy assertion.

…but again, following the Service Loose Coupling principle (as well as the Service Abstraction principle), implementation details should really be kept private.

There will certainly be less controversial usages for this attribute for which you may be tempted to express configuration, environmental, or deployment characteristics of a Web service.

The number one question you need to raise when considering any of these types of assertions is: "What are the governance implications?" In other words, you need to weigh how useful an ignorable assertion may be to consumers against the impact of having to maintain and perhaps change this assertion in the future.

NOTE

When maintaining policies using proprietary vendor platform features, you may be required to create various types of ignorable assertions (or these assertions may be created for you automatically by the vendor tools) that *do* express implementation details. This is especially the case when most consumers (which, of course, can also be Web services) are being built and deployed in the same vendor environment. Be sure to assess the long-term impact of using proprietary features before committing to them.

SUMMARY OF KEY POINTS

- A policy assertion can be tagged as "ignorable" in order to communicate that the assertion that will be in effect does not need to be acknowledged by the consumer.
- Ignorable assertions are primarily used for information purposes to convey to consumers that a certain type of processing will occur on the service-side, regardless of whether the consumer performs any special processing in response to the assertion.
- Ignorable assertions are different from optional assertions and the `wsp:Ignorable` attribute is almost never used together with the `wsp:Optional` attribute.

16.4 Concurrent Policy-Enabled Contracts

The Concurrent Contract design pattern provides a design option whereby the same underlying service logic can expose two or more different contracts. When services are built as Web services, this pattern is more easily implemented because of the fact that Web service contracts are physically decoupled from their underlying implementation.

When designing policy alternatives for Web service contracts, you can end up with some very complex and elaborate structures. While these may be justified, it is worth understanding that you can also apply the Concurrent Contracts pattern as a means of establishing multiple Web service contracts that each express a policy expression (or perhaps a subset of the overall policy alternatives). In other words, creating multiple contracts may be a viable alternative to policy alternatives.

One reason in particular to consider this approach is when a Web service needs to accommodate both trusted and non-trusted consumer programs. You, as the service owner, may not want to expose a detailed set of policy alternatives to the non-trusted

Erl_16.qxd   8/7/08   9:55 AM   Page 29

consumers because some of the alternative policy expressions may include private or business-related assertions that should only be made available to trusted parties.

NOTE

You can consider the approach of having concurrent policies as a specialized implementation of the Concurrent Contracts design pattern.

In the *Case Study Example: Split Concrete Descriptions Sharing the Same Abstract Description* section from Chapter 14, we demonstrated how two different WSDL concrete descriptions each imported the same abstract description for reuse purposes.

Let's revisit this example now to add a different policy expression to each concrete description:

The concrete description for the SOAP 1.1 binding with an ignorable assertion:

```
<definitions targetNamespace=
  "http://actioncon.com/contract/POSoap11"
  xmlns:tns="http://actioncon.com/contract/POSoap11"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
  xmlns:abs="http://actioncon.com/contract/po"
  xmlns:po="http://actioncon.com/schema/purchasing"
  xmlns:custom="http://actioncon.com/policy/custom"
  xmlns:soap11="http://schemas.xmlsoap.org/wsdl/soap/">
  <import
    namespace="http://actioncon.com/contract/PurchaseOrder"
    location="http://actioncon.com/contract/PurchaseOrder"/>
  <binding name="bdPO-SOAP11HTTP" type="abs:ptPurchaseOrder">
  <soap11:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="opSubmitOrder">
      <input><soap11:body use="literal"/></input>
      <output><soap11:body use="literal"/></output>
    </operation>
    <operation name="opCheckOrderStatus">
      <input><soap11:body use="literal"/></input>
      <output><soap11:body use="literal"/></output>
    </operation>
    <operation name="opChangeOrder">
      <input><soap11:body use="literal"/></input>
      <output><soap11:body use="literal"/></output>
```

Erl_16.qxd 8/7/08 9:55 AM Page 30

```
    </operation>
    <operation name="opCancelOrder">
      <input><soap11:body use="literal"/></input>
      <output><soap11:body use="literal"/></output>
    </operation>
  </binding>
  <service name="svPurchaseOrder">
    <wsp:Policy>
      <custom:LogMessage wsp:Ingorable="true"/>
    </wsp:Policy>
    <port name="purchaseOrder-soap11http"
      binding="tns:bdPO-SOAP11HTTP">
      <soap11:address location=
        "http://actioncon.com/services/soap11/purchaseOrder"/>
    </port>
  </service>
</definitions>
```

Example 16.25
A WSDL definition with an embedded, ignorable policy assertion.

The concrete description for the SOAP 1.2 binding with one required and one ignorable assertion:

```
<definitions targetNamespace=
  "http://actioncon.com/contract/POSoap12"
  xmlns:tns="http://actioncon.com/contract/POSoap12"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
  xmlns:wsam=" http://www.w3.org/2007/05/addressing/
    metadata"
  xmlns:custom="http://actioncon.com/policy/custom"
  xmlns:abs="http://actioncon.com/contract/po"
  xmlns:po="http://actioncon.com/schema/purchasing"
  xmlns:soap11="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
  <import
     namespace="http://actioncon.com/contract/PurchaseOrder"
     location="http://actioncon.com/contract/PurchaseOrder"/>
  <binding name="bdPO-SOAP12HTTP" type="abs:ptPurchaseOrder">
  <soap12:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="opSubmitOrder">
      <soap12:operation soapAction=
        "http://actioncon.com/submitOrder/request"
        soapActionRequired="true" required="true"/>
```

Erl_16.qxd 8/7/08 9:55 AM Page 31

```
        <input><soap12:body use="literal"/></input>
        <output><soap12:body use="literal"/></output>
    </operation>
    <operation name="opCheckOrderStatus">
    <soap12:operation soapAction=
      "http://actioncon.com/submitOrder/request"
        soapActionRequired="true" required="true"/>
      <input><soap12:body use="literal"/></input>
      <output><soap12:body use="literal"/></output>
    </operation>
    <operation name="opChangeOrder">
    <soap12:operation soapAction=
      "http://actioncon.com/submitOrder/request"
      soapActionRequired="true" required="true"/>
      <input><soap12:body use="literal"/></input>
      <output><soap12:body use="literal"/></output>
    </operation>
    <operation name="opCancelOrder">
    <soap12:operation soapAction=
      "http://actioncon.com/submitOrder/request"
      soapActionRequired="true" required="true"/>
      <input><soap12:body use="literal"/></input>
      <output><soap12:body use="literal"/></output>
    </operation>
  </binding>
  <service name="svPurchaseOrder">
    <wsp:Policy>
      <wsp:All>
        <wsam:Addressing/>
        <custom:LogMessage wsp:Ingorable="true"/>
      </wsp:All>
    </wsp:Policy>
    <port name="purchaseOrder-soap11http"
      binding="tns:bdPO-SOAP11HTTP">
      <soap11:address location=
        "http://actioncon.com/services/soap11/purchaseOrder"/>
    </port>
    <port name="purchaseOrder-http-soap12"
      binding="tns:bdPO-SOAP12HTTP">
      <soap12:address location=
        "http://actioncon.com/services/soap12/purchaseOrder"/>
    </port>
  </service>
</definitions>
```

Example 16.26

A different version of the WSDL definition with an embedded policy expression comprised of required and ignorable policy assertions.

You might recall that the two versions of the concrete descriptions were created to accommodate different types of consumers. Those that support SOAP 1.2 are now also being asked to support WS-Addressing headers as per the new `wsam:Addressing` assertion.

The alternate concrete description that exposes operations via SOAP 1.1 is intended for external partner organizations with less progressive technology platforms. Therefore, the `custom:LogMessage` assertion is added for informational purposes only. Any consumers incapable of working with WS-Policy will not be affected by the presence of this assertion because it is ignorable.

SUMMARY OF KEY POINTS

- When the Concurrent Contracts pattern is applied to a Web service, it provides a design option whereby a single body of service logic is exposed via multiple Web service contracts.
- Instead of creating elaborate policy alternatives, you can consider applying this pattern to provide alternate Web service contracts where each contains a different policy expression or a different set of policy alternatives.
- This technique is most commonly used to accommodate different groups of consumers and also to release alternate versions of Web service contracts for security reasons.