

# The SOA Magazine

## Feature Article



### Understanding WS-Policy Part I: Policy Structure and Composite Policies

by Umit Yalcinalp and Thomas Erl

Published: February 23, 2009 (SOA Magazine Issue XXVI: February 2009)

[Download PDF](#)

[Digg This](#) • [De.licio.us](#) • [Slashdot](#) • [Technorati](#) • [StumbleUpon](#) • [Google Bookmark](#)

*Abstract: Web service contracts can be extended with policies that express additional constraints, requirements, and qualities that typically relate to the behaviors of services. You can create human-readable policies that become part of a supplemental service-level agreement, or you can define machine-readable policies that are processed at runtime. The latter type of policy is the focus of this article and the technology we'll explore to create machine-readable policies is the WS-Policy language and related WS-Policy specifications.*

*The following article is an excerpt from the new book "Web Service Contract Design and Versioning for SOA" [REF-1] Copyright Prentice Hall/Pearson PTR and SOA Systems Inc. Note that some chapter references were intentionally left in the article, as per requirements from Prentice Hall.*

### Introduction

There are many kinds of policies, some of which are pre-defined by industry specifications and others that can be customized by the Web service contract designer. For example, you can use policies to express interoperability and protocol constraints, as well as privacy, manageability, and quality of service (QoS) requirements. Furthermore, policies may be created for a single party, or they may apply to multiparty interactions.

From an architectural and contract design perspective, policies affect all levels of a service design. They can be applied to most parts of the abstract and concrete descriptions of a WSDL document, and they can be applied at different scopes. For example, one policy may pertain to a message definition while another is applied to the entire WSDL port type (or interface).

In this article, we first cover the essential aspects of policies and then introduce the fundamental parts of the WS-Policy language. Chapters 16 and 17 continue the exploration of the WS-Policy framework by documenting a range of advanced language features and design techniques.

The XML Schema and WSDL languages allow us to do little more than express the interaction requirements and constraints of a Web service. While this is, of course, fundamentally important for us to do anything useful with the Web service, it does not provide us with the opportunity to describe other aspects of the Web service, such as:

- Are there certain QoS requirements (reliability, security, etc.) that consumer programs will need to adhere to in order to work with the service?
- Are there any additional requirements as to how the service can or cannot be accessed?
- Are there properties or characteristics of the service that might be of interest to consumer programs?
- Are there certain rules that must be followed in order to interact with the service?

## Policy Structure

The WS-Policy vocabulary is relatively simple in comparison to WSDL and XML Schema in that it contains only a modest amount of elements and attributes. However, policies introduce unique structural considerations that differ from the straightforward technical interface focus of WSDL and XML Schema. Because policies are about expressing behavioral qualities, they can range dramatically in size and in the nature of the policy content. Additionally, the flexibility and extensibility built in to the WS-Policy language allows its few elements and attributes to be combined into a variety of complex designs.

Before we learn more about the individual WS-Policy language elements, let's first introduce some basic terminology:

- The formal term for a policy is *policy expression*.
- A policy expression can be comprised of one or more elements that express specific policy requirements or properties. Each of these is called a *policy assertion*.
- In order to group policy assertions, we use a set of features from the WS-Policy language known as *policy operators*.
- Policy expressions can optionally be isolated into a separate document, referred to as a *WS-Policy definition*.

Among all of these parts of the WS-Policy framework, the most basic building block is the policy assertion.

## New Namespaces and Prefixes

Let's now take a minute to establish some new namespaces and prefixes associated with the WS-Policy language and common policies:

- `xmlns:wsp="http://www.w3.org/2006/07/ws-policy"` - This represents the actual namespace used for elements from the WS-Policy language.
- `xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"` - We will show some policy examples related to the WS-Addressing language, which is why this namespace comes up here. How the referenced WS-Addressing features actually work is covered in Chapter 18 and this particular policy assertion is further revisited at the end of Chapter 19.
- `xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702"` - This namespace corresponds to the WS-ReliableMessaging policy assertion. Although WS-ReliableMessaging is not a technology covered in this book, there are a few references to one of its policy assertions in the examples.
- `xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"` - There is a special type of schema that is referred to as a "utility schema" in which generic and commonly used attributes are established. One such attribute is `wsu:Id`, a simple ID used to associate an identifier with an element. This and other chapters make occasional reference to this attribute.

Note that throughout the examples in we generally avoid displaying the previously listed `xmlns` values simply to avoid repeated clutter of policy definition code fragments.

## Assertions, Expressions, and the Policy Element

The most important thing to understand about WS-Policy is that it is not a technology or language used to enforce policies. Its sole purpose is to provide a standardized syntax with which to express policies (hence the term "policy expression").

### Policy Assertions

It all begins with a policy assertion, which, on its own, is simply a reference to a predefined XML Schema global element.

Here's an example of a simple policy assertion:

```
<wsrmp:RMAssertion>>wsp:Policy/>>/wsrmp:RMAssertion>
```

The `wsrmp:RMAssertion` element refers to an extensibility element defined in the WS-ReliableMessaging specification. This is therefore considered a WS-ReliableMessaging policy assertion.

The XML schema that defines this assertion looks like this:

```
<xsd:schema
  xmlns:tns="http://docs.oasis-open.org/ws-rx/wsrmp/200608"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://docs.oasisopen.org/wsrx/wsrmp/200608"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
  <xsd:element name="RMAssertion">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:any namespace="##other" processContents="lax"
          minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:anyAttribute namespace="##any"
        processContents="lax"/>
    </xsd:complexType>
  </xsd:element>
  ...
</xsd:schema>
```

We refer to what's defined in this schema as the *policy type*. Every assertion has an underlying policy type, which can be provided by an industry standard like WS-Reliable-Messaging or you can custom define it by creating your own schema. All you need is a global element that can be referenced in the policy assertion syntax.

Beyond providing a standardized policy expression, WS-Policy has nothing to do with how policy assertions are actually processed. For example, what happens when the `wsrmp:RMAssertion` element is encountered at runtime is up to WS-ReliableMessaging related processors. All we use WS-Policy for is to associate this assertion with a WSDL definition.

Similarly, if you create your own custom policy assertions, you will need to provide the underlying processing logic that does something in response to encountering the policy assertion being processed at runtime.

We're going to be using the `wsrmp:RMAssertion` element on and off throughout the WS-Policy chapters. Let's now take a look at another example:

```
<wsam:Addressing><wsp:Policy/></wsam:Addressing>
```

WS-Addressing provides a pre-defined `wsam:Addressing` extensibility element that can be used in the WSDL binding construct to associate WS-Addressing with the Web service.

As with `wsrmp:RMAssertion`, this assertion introduces the requirement that incoming messages support a particular technology; in this case, WS-Addressing.

### Policy Expressions

On their own, the preceding policy assertion statements do not yet exist as standalone policies. In order create an actual *policy expression*, we need to wrap assertions in a `wsp:Policy` construct, as follows:

```
<wsp:Policy>
  <wsam:Addressing><wsp:Policy/></wsam:Addressing>
</wsp:Policy>
```

```

<wsp:Policy>
  <wsrmp:RMAssertion><wsp:Policy/></wsrmp:RMAssertion>
</wsp:Policy>
<wsp:Policy>
  <wsam:Addressing><wsp:Policy/></wsam:Addressing>
  <wsrmp:RMAssertion><wsp:Policy/></wsrmp:RMAssertion>
</wsp:Policy>

```

### Custom Policy Assertions

Let's now assume you want to create your own policy expression with an assertion that defines a response guarantee for a particular operation. Let's first create the XML Schema element that forms the basis of the required policy assertion.

```

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/respguarantee"
  elementFormDefault="qualified">
  <xsd:element name="responseGuarantee"
    type="ResponseGuaranteeType"/>
  <xsd:complexType name="ResponseGuaranteeType">
    <xsd:sequence>
      <xsd:element name="responseInMilliseconds"
        type="xsd:integer"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

```

This is a good example of a custom-created QoS policy that can be added to a service contract so that consumers could retrieve the number of milliseconds within which the service promises to perform a particular task.

If you recall the ActionCon Purchase Order service, an assertion such as this could be used to indicate the amount of time the service will take to generate and return a purchase order document. Depending on the circumstances, the actual value of the assertion may differ, as shown in the following two assertion instances.

#### Assertion Instance #1

```

<argt:responseGuarantee>
  <argt:responseInMilliseconds>
    50
  </argt:responseInMilliseconds>
</argt:responseGuarantee>

```

#### Assertion Instance #2

```

<argt:responseGuarantee>
  <argt:responseInMilliseconds>on.
    30
  </argt:responseInMilliseconds>
</argt:responseGuarantee>

```

Both assertions have the same assertion element, namely `argt:responseGuarantee`. However, each instance has a different value.

### Composite Policies

In the previous example we created a simple policy assertion. On its own, this one assertion may be sufficient to warrant an entire, self-contained policy expression. However, more often than not, policies need to be assembled out of multiple policy assertions.

In order to combine policy assertion constructs into composite policy expressions, we need to use a new feature of the WS-Policy language known as *operators*. Specifically, we will need to work with the `wsp:ExactlyOne` and `wsp:All` elements and the `wsp:optional` attribute.

### The ExactlyOne Element

This element groups a set of policy assertions from which only one can be used. In other words, it enforces a rule that "exactly one" of the listed assertions must be chosen.

Using the `wsp:ExactlyOne` element introduces the concept of *policy alternatives* as part of a policy expression. Each child element within this construct is considered a distinct alternative in the overall policy expression.

Consider the following example:

```
<wsp:Policy>
  <wsp:ExactlyOne>
    <wsam:Addressing><wsp:Policy/></wsam:Addressing>
    <wsrmp:RMAssertion><wsp:Policy/></wsrmp:RMAssertion>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Here, the WS-Policy expression contains two distinct alternatives. Each assertion, `wsam:Addressing` and `wsrmp:RMAssertion`, indicates a separate alternative.

### The wsp:All Element

The `wsp:All` operator element is the reverse of the `wsp:ExactlyOne` element. It imposes a rule that requires that all of the assertions within its construct be applied at the same time.

If we restructure the previous example using this element, it actually looks quite similar.

```
<wsp:Policy>
  <wsp:All>
    <wsam:Addressing><wsp:Policy/></wsam:Addressing>
    <wsrmp:RMAssertion><wsp:Policy/></wsrmp:RMAssertion>
  </wsp:All>
</wsp:Policy>>
```

As with the `wsp:ExactlyOne` construct, it does not matter in what order you decide to place the policy assertion type elements.

### The wsp:optional Attribute

The WS-Policy language provides a special Boolean attribute named `wsp:optional` that allows you to indicate that a policy assertion is not mandatory.

The following example shows this attribute used in conjunction with our previous `wsam:Addressing` assertion:

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

This attribute is considered a "shortcut" because you can express the same policy logic in a more verbose manner using the previously described operator elements. The `wsp:All` element is essentially the same as specifying two distinct alternatives in a policy expression, as follows:

```
<wsp:Policy>
  <wsp:ExactlyOne>
    <wsp:All>
      <wsam:Addressing><wsp:Policy/></wsam:Addressing>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

In this example, we have a `wsp:ExactlyOne` construct that houses two `wsp:All` elements. The second `wsp:All` element is highlighted. Unlike the first, which establishes a construct with one assertion, this second `wsp:All` element is empty.

According to the "exactly one" rule, we must choose one of the two alternatives. The first alternative is that the `wsam:Addressing` assertion is used because in this alternative, the assertion is wrapped in a nested `wsp:All` construct. The second alternative is that the second `wsp:All` element be chosen, and because it is empty, the result of this choice is that nothing happens. In other words, the use of the policy assertion is optional, as with the previous example that used the `wsp:optional` attribute. This last example also hints at the more complex structures that can be built using nested operator elements and various other combinations, as described further in the next section.

## Conclusion

Part II of this article series will focus on operator composition rules and techniques for attaching policies to WSDL documents.

## References

[REF-1] "Web Service Contract Design and Versioning for SOA" by Thomas Erl, Anish Karmarkar, Priscilla Walmsley, Hugo Haas, Umit Yalcinalp, Canyon Kevin Liu, David Orchard, Andre Tost, James Pasley for the "Prentice Hall Service-Oriented Computing Series from Thomas Erl", Copyright Prentice Hall/Pearson PTR and SOA Systems Inc., <http://www.soabooks.com/wsc/>

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL

