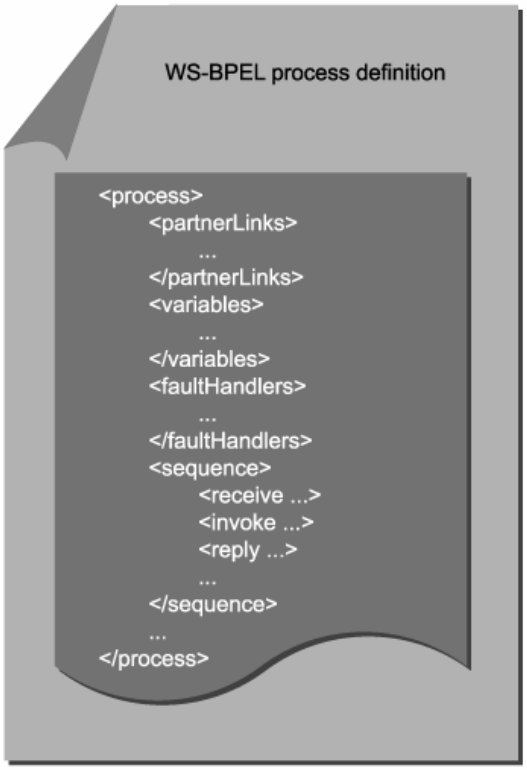
UNIT - V

WS-BPEL language basics



Although you likely will be using a process modeling tool and will therefore not be required to author your process definition from scratch, a knowledge of WS-BPEL elements still is useful and often required. WS-BPEL modeling tools frequently make reference to these elements and constructs, and you may be required to dig into the source code they produce to make further refinements.

A brief history of BPEL4WS and WS-BPEL

Before we get into the details of the WS-BPEL language, let's briefly discuss how this specification came to be. The Business Process Execution Language for Web Services (BPEL4WS) was first conceived in July, 2002, with the release of the BPEL4WS 1.0 specification, a joint effort by IBM, Microsoft, and BEA. This document proposed an orchestration language inspired by previous variations, such as IBM's Web Services Flow Language (WSFL) and Microsoft's XLANG specification.

Joined by other contributors from SAP and Siebel Systems, version 1.1 of the BPEL4WS specification was released less than a year later, in May of 2003. This version received more attention and vendor support, leading to a number of commercially available BPEL4WS-compliant orchestration engines. Just prior to this release, the BPEL4WS specification was submitted to an OASIS technical committee so that the specification could be developed into an

official, open standard.

The `process` element

Let's begin with the root element of a WS-BPEL process definition. It is assigned a name value using the `name` attribute and is used to establish the process definition-related namespaces.

Example 16.1. A skeleton `process` definition.

```
<process name="TimesheetSubmissionProcess"
   targetNamespace="http://www.xmltc.com/tls/process/"
   xmlns=
      "http://schemas.xmlsoap.org/ws/2003/03/
          business-process/"
   xmlns:bpl="http://www.xmltc.com/tls/process/"
   xmlns:emp="http://www.xmltc.com/tls/employee/"
   xmlns:inv="http://www.xmltc.com/tls/invoice/"
   xmlns:tst="http://www.xmltc.com/tls/timesheet/"
   xmlns:not="http://www.xmltc.com/tls/notification/">
   <partnerLinks>
      ...
   </partnerLinks>
   <variables>
      ...
   </variables>
   <sequence>
      ...
   </sequence>
   ...
</process>
```

The `process` construct contains a series of common child elements explained in the following sections.

The `partnerLinks` and `partnerLink` elements

A `partnerLink` element establishes the port type of the service (partner) that will be participating during the execution of the business process. Partner services can act as a client to the process, responsible for invoking the process service. Alternatively, partner services can be invoked by the process service itself.

The contents of a `partnerLink` element represent the communication exchange between two partnersthe process service being one partner and another service being the other. Depending on the nature of the communication, the role of the process service will vary. For instance, a process service that is invoked by an external service may act in the role of "TimesheetSubmissionProcess." However, when this same process service invokes a different service to have an invoice verified, it acts within a different role, perhaps "InvoiceClient." The `partnerLink` element therefore contains the `myRole` and `partnerRole` attributes that establish the service provider role of the process service and the partner service respectively.

Put simply, the `myRole` attribute is used when the process service is invoked by a partner client

service, because in this situation the process service acts as the service provider. The `partnerRole` attribute identifies the partner service that the process service will be invoking (making the partner service the service provider).

Note that both `myRole` and `partnerRole` attributes can be used by the same `partnerLink` element when it is expected that the process service will act as both service requestor and service provider with the same partner service. For example, during asynchronous communication between the process and partner services, the `myRole` setting indicates the process service's role during the callback of the partner service.

The `partnerLinks` construct containing one `partnerLink` element in which the process service is invoked by an external client partner and four `partnerLink` elements that identify partner services invoked by the process service.

```
<partnerLinks>
   <partnerLink name="client"
      partnerLinkType="tns:TimesheetSubmissionType"
      myRole="TimesheetSubmissionServiceProvider"/>
   <partnerLink name="Invoice"
      partnerLinkType="inv:InvoiceType"
      partnerRole="InvoiceServiceProvider"/>
   <partnerLink name="Timesheet"
      partnerLinkType="tst:TimesheetType"
      partnerRole="TimesheetServiceProvider"/>
   <partnerLink name="Employee"
      partnerLinkType="emp:EmployeeType"
      partnerRole="EmployeeServiceProvider"/>
   <partnerLink name="Notification"
      partnerLinkType="not:NotificationType"
      partnerRole="NotificationServiceProvider"/>
</partnerLinks>
```

The `partnerLinkType` element

For each partner service involved in a process, `partnerLinkType` elements identify the WSDL `portType` elements referenced by the `partnerLink` elements within the process definition. Therefore, these constructs typically are embedded directly within the WSDL documents of every partner service (including the process service).

The `partnerLinkType` construct contains one role element for each role the service can play, as defined by the `partnerLink` `myRole` and `partnerRole` attributes. As a result, a `partnerLinkType` will have either one or two child role elements.

A WSDL `definitions` construct containing a `partnerLinkType` construct.

```
<definitions name="Employee"
   targetNamespace="http://www.xmltc.com/tls/employee/wsdl/"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:plnk=
      "http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
   ...
>
   ...
```

```
   <plnk:partnerLinkType name="EmployeeServiceType" xmlns=
      "http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
      <plnk:role name="EmployeeServiceProvider">
         <portType name="emp:EmployeeInterface"/>
      </plnk:role>
   </plnk:partnerLinkType>
   ...
</definitions>
```

Note that multiple `partnerLink` elements can reference the same `partnerLinkType`. This is useful for when a process service has the same relationship with multiple partner services. All of the partner services can therefore use the same process service `portType` elements.

Note

In version 2.0 of the WS-BPEL specification, it is being proposed that the `portType` element be changed so that it exists as an attribute of the `role` element.

The `variables` element

WS-BPEL process services commonly use the `variables` construct to store state information related to the immediate workflow logic. Entire messages and data sets formatted as XSD schema types can be placed into a variable and retrieved later during the course of the process. The type of data that can be assigned to a `variable` element needs to be predefined using one of the following three attributes: `messageType`, `element`, or `type`.

The `messageType` attribute allows for the variable to contain an entire WSDL-defined message, whereas the `element` attribute simply refers to an XSD element construct. The `type` attribute can be used to just represent an XSD `simpleType`, such as string or integer.

The `variables` construct hosting only some of the child `variable` elements used later by the Timesheet Submission Process.

```
<variables>
   <variable name="ClientSubmission"
      messageType="bpl:receiveSubmitMessage"/>
   <variable name="EmployeeHoursRequest"
      messageType="emp:getWeeklyHoursRequestMessage"/>
   <variable name="EmployeeHoursResponse"
      messageType="emp:getWeeklyHoursResponseMessage"/>
   <variable name="EmployeeHistoryRequest"
      messageType="emp:updateHistoryRequestMessage"/>
   <variable name="EmployeeHistoryResponse"
      messageType="emp:updateHistoryResponseMessage"/>
   ...
</variables>
```

Typically, a variable with the `messageType` attribute is defined for each input and output

message processed by the process definition. The value of this attribute is the message name from the partner process definition.

The `getVariableProperty` and `getVariableData` functions

WS-BPEL provides built-in functions that allow information stored in or associated with variables to be processed during the execution of a business process.

getVariableProperty(variable name, property name)

This function allows global property values to be retrieved from variables. It simply accepts the variable and property names as input and returns the requested value.

getVariableData(variable name, part name, location path)

Because variables commonly are used to manage state information, this function is required to provide other parts of the process logic access to this data. The `getVariableData` function has a mandatory variable name parameter and two optional arguments that can be used to specify a part of the variable data.

In our examples we use the `getVariableData` function a number of times to retrieve message data from variables.

Two `getVariableData` functions being used to retrieve specific pieces of data from different variables.

```
getVariableData ('InvoiceHoursResponse',
                 'ResponseParameter')

getVariableData ('input','payload',
                 '/tns:TimesheetType/Hours/...')
```

The `sequence` element

The `sequence` construct allows you to organize a series of activities so that they are executed in a predefined, sequential order. WS-BPEL provides numerous activities that can be used to express the workflow logic within the process definition. The remaining element descriptions in this section explain the fundamental set of activities used as part of our upcoming case study examples.

A skeleton `sequence` construct containing only some of the many activity elements provided by WS-BPEL.

```
<sequence>
   <receive>
      ...
   </receive>
   <assign>
      ...
   </assign>
```

```
   <invoke>
      ...
   </invoke>
   <reply>
      ...
   </reply>
</sequence>
```

The `invoke` element

This element identifies the operation of a partner service that the process definition intends to invoke during the course of its execution. The `invoke` element is equipped with five common attributes, which further specify the details of the invocation

`invoke` element attributes

Attribute	Description
`partnerLink`	This element names the partner service via its corresponding `partnerLink`.
`portType`	The element used to identify the `portType` element of the partner service.
`operation`	The partner service operation to which the process service will need to send its request.
`inputVariable`	The input message that will be used to communicate with the partner service operation. Note that it is referred to as a variable because it is referencing a WS-BPEL `variable` element with a `messageType` attribute.
`outputVariable`	This element is used when communication is based on the request-response MEP. The return value is stored in a separate `variable` element.

The `invoke` element identifying the target partner service details.

```
<invoke name="ValidateWeeklyHours"
   partnerLink="Employee"
   portType="emp:EmployeeInterface"
   operation="GetWeeklyHoursLimit"
   inputVariable="EmployeeHoursRequest"
   outputVariable="EmployeeHoursResponse"/>
```

The `receive` element

The `receive` element allows us to establish the information a process service expects upon receiving a request from an external client partner service. In this case, the process service is viewed as a service provider waiting to be invoked.

The `receive` element contains a set of attributes, each of which is assigned a value relating to the expected incoming communication

`receive` element attributes

Attribute	Description
`partnerLink`	The client partner service identified in the corresponding `partnerLink` construct.
`portType`	The process service `portType` that will be waiting to receive the request message from the partner service.
`operation`	The process service operation that will be receiving the request.
`variable`	The process definition `variable` construct in which the incoming request message will be stored.
`createInstance`	When this attribute is set to "yes," the receipt of this particular request may be responsible for creating a new instance of the process.

The `receive` element used in the Timesheet Submission Process definition to indicate the client partner service responsible for launching the process with the submission of a timesheet document.

```
<receive name="receiveInput"
   partnerLink="client"
   portType="tns:TimesheetSubmissionInterface"
   operation="Submit"
   variable="ClientSubmission"
   createInstance="yes"/>
```

The `reply` element

Where there's a `receive` element, there's a `reply` element when a synchronous exchange is being mapped out. The `reply` element is responsible for establishing the details of returning a response message to the requesting client partner service. Because this element is associated with the same `partnerLink` element as its corresponding `receive` element, it repeats a number of the same attributes

`reply` element attributes

Attribute	Description
`partnerLink`	The same `partnerLink` element established in the `receive` element.
`portType`	The same `portType` element displayed in the `receive` element.
`operation`	The same `operation` element from the `receive` element.
`variable`	The process service `variable` element that holds the message that is returned to the partner service.
`messageExchange`	It is being proposed that this optional attribute be added by the WS-BPEL

	2.0 specification. It allows for the `reply` element to be explicitly associated with a message activity capable of receiving a message (such as the `receive` element).

A potential companion `reply` element to the previously displayed `receive` element.

```
<reply partnerLink="client"
   portType="tns:TimesheetSubmissionInterface"
   operation="Submit"
   variable="TimesheetSubmissionResponse"/>
```

The `switch`, `case`, and `otherwise` elements

These three structured activity elements allow us to add conditional logic to our process definition, similar to the familiar select case/case else constructs used in traditional programming languages. The `switch` element establishes the scope of the conditional logic, wherein multiple `case` constructs can be nested to check for various conditions using a `condition` attribute. When a `condition` attribute resolves to "true," the activities defined within the corresponding `case` construct are executed.

The `otherwise` element can be added as a catch all at the end of the `switch` construct. Should all preceding `case` conditions fail, the activities within the `otherwise` construct are executed.

A skeleton `case` element wherein the `condition` attribute uses the `getVariableData` function to compare the content of the EmployeeResponseMessage variable to a zero value.

```
<switch>
   <case condition=
      "getVariableData('EmployeeResponseMessage',
                       'ResponseParameter')=0">
      ...
   </case>
   <otherwise>
      ...
   </otherwise>
</switch>
```

The `assign`, `copy`, `from`, and `to` elements

This set of elements simply gives us the ability to copy values between process variables, which allows us to pass around data throughout a process as information is received and modified during the process execution.

Within this `assign` construct, the contents of the `TimesheetSubmissionFailedMessage` variable are copied to two different message variables.

```
<assign>
   <copy>
      <from variable="TimesheetSubmissionFailedMessage"/>
      <to variable="EmployeeNotificationMessage"/>
   </copy>
```

```
   <copy>
      <from variable="TimesheetSubmissionFailedMessage"/>
      <to variable="ManagerNotificationMessage"/>
   </copy>
</assign>
```

`faultHandlers`, `catch`, and `catchAll` elements

This construct can contain multiple `catch` elements, each of which provides activities that perform exception handling for a specific type of error condition. Faults can be generated by the receipt of a WSDL-defined fault message, or they can be explicitly triggered through the use of the `throw` element. The `faultHandlers` construct can consist of (or end with) a `catchAll` element to house default error handling activities.

The `faultHandlers` construct hosting `catch` and `catchAll` child constructs.

```
<faultHandlers>
   <catch faultName="SomethingBadHappened"
          faultVariable="TimesheetFault">
      ...
   </catch>
   <catchAll>
      ...
   </catchAll>
</faultHandlers>
```

Other WS-BPEL elements

The following table provides brief descriptions of other relevant parts of the WS-BPEL language.

Quick reference table providing short descriptions for additional WS-BPEL elements (listed in alphabetical order).

Element	Description
`compensationHandler`	A WS-BPEL process definition can define a compensation process that kicks in a series of activities when certain conditions occur to justify a compensation. These activities are kept in the `compensationHandler` construct. (For more information about compensations, see the Business activities section in Chapter 6.)
`correlationSets`	WS-BPEL uses this element to implement correlation, primarily to associate messages with process instances. A message can belong to multiple `correlationSets`. Further, message properties can be defined within WSDL documents.
`empty`	This simple element allows you to state that no activity should occur for a particular condition.
`eventHandlers`	The `eventHandlers` element enables a process to respond to events during the execution of process logic. This construct can contain

	`onMessage` and `onAlarm` child elements that trigger process activity upon the arrival of specific types of messages (after a predefined period of time, or at a specific date and time, respectively).
`exit`	See the `terminate` element description that follows.
`flow`	A `flow` construct allows you to define a series of activities that can occur concurrently and are required to complete after all have finished executing. Dependencies between activities within a `flow` construct are defined using the child `link` element.
`pick`	Similar to the `eventHandlers` element, this construct also can contain child `onMessage` and `onAlarm` elements but is used more to respond to external events for which process execution is suspended.
`scope`	Portions of logic within a process definition can be sub-divided into scopes using this construct. This allows you to define `variables`, `faultHandlers`, `correlationSets`, `compensationHandler`, and `eventHandlers` elements local to the scope.
`terminate`	This element effectively destroys the process instance. The WS-BPEL 2.0 specification proposes that this element be renamed `exit`.
`throw`	WS-BPEL supports numerous fault conditions. Using the `tHRow` element allows you to explicitly trigger a fault state in response to a specific condition.
`wait`	The `wait` element can be set to introduce an intentional delay within the process. Its value can be a set time or a predefined date.
`while`	This useful element allows you to define a loop. As with the `case` element, it contains a `condition` attribute that, as long as it continues resolving to "true," will continue to execute the activities within the `while` construct.

WS-Coordination overview

The `CoordinationContext` element

This parent construct contains a series of child elements that each house a specific part of the context information being relayed by the header.

A skeleton `CoordinationContext` construct.

```
<Envelope
   xmlns="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:wsc=
      "http://schemas.xmlsoap.org/ws/2002/08/wscoor"
   xmlns:wsu=
      "http://schemas.xmlsoap.org/ws/2002/07/utility">
   <Header>
      <wsc:CoordinationContext>
         <wsu:Identifier>
            ...
         </wsu:Identifier>
         <wsu:Expires>
            ...
         </wsu:Expires>
         <wsc:CoordinationType>
            ...
         </wsc:CoordinationType>
         <wsc:RegistrationService>
            ...
         </wsc:RegistrationService>
      </wsc:CoordinationContext>
   </Header>
   <Body>
      ...
   </Body>
</Envelope>
```

The activation service returns this `CoordinationContext` header upon the creation of a new activity. As described later, it is within the `CoordinationType` child construct that the activity protocol (WS-BusinessActivity, WS-AtomicTransaction) is carried. Vendor-specific implementations of WS-Coordination can insert additional elements within the `CoordinationContext` construct that represent values related to the execution environment.

The `Identifier` and `Expires` elements

These two elements originate from a utility schema used to provide reusable elements. WS-Coordination uses the `Identifier` element to associate a unique ID value with the current activity. The `Expires` element sets an expiry date that establishes the extent of the activity's possible lifespan.

`Identifier` and `Expires` elements containing values relating to the header.

```
<Envelope
   ...
   xmlns:wsu=
      "http://schemas.xmlsoap.org/ws/2002/07/utility">
   ...
   <wsu:Identifier>
      http://www.xmltc.com/ids/process/33342
   </wsu:Identifier>
   <wsu:Expires>
      2008-07-30T24:00:00.000
   </wsu:Expires>
   ...
</Envelope>
```

The `CoordinationType` element

This element is described shortly in the WS-BusinessActivity and WS-AtomicTransaction coordination types section.

The `RegistrationService` element

The `RegistrationService` construct simply hosts the endpoint address of the registration service. It uses the `Address` element also provided by the utility schema.

The `RegistrationService` element containing a URL pointing to the location of the registration service.

```
<wsc:RegistrationService>
   <wsu:Address>
      http://www.xmltc.com/bpel/reg
   </wsu:Address>
</wsc:RegistrationService>
```

Designating the WS-BusinessActivity coordination type

The specific protocol(s) that establishes the rules and constraints of the activity are identified within the `CoordinationType` element. The URI values that are placed here are predefined within the WS-BusinessActivity and WS-AtomicTransaction specifications.

This first example shows the `CoordinationType` element containing the WS-BusinessActivity coordination type identifier. This would indicate that the activity for which the header is carrying context information is a potentially long-running activity.

The `CoordinationType` element representing the WS-BusinessActivity protocol.

```
<wsc:CoordinationType>
   http://schemas.xmlsoap.org/ws/2004/01/wsba
</wsc:CoordinationType>
```

Designating the WS-AtomicTransaction coordination type

In the next example, the `CoordinationType` element is assigned the WS-AtomicTransaction coordination type identifier, which communicates the fact that the header's context information is part of a short running transaction.

The `CoordinationType` element representing the WS-AtomicTransaction protocol.

```
<wsc:CoordinationType>
   http://schemas.xmlsoap.org/ws/2003/09/wsat
</wsc:CoordinationType>
```

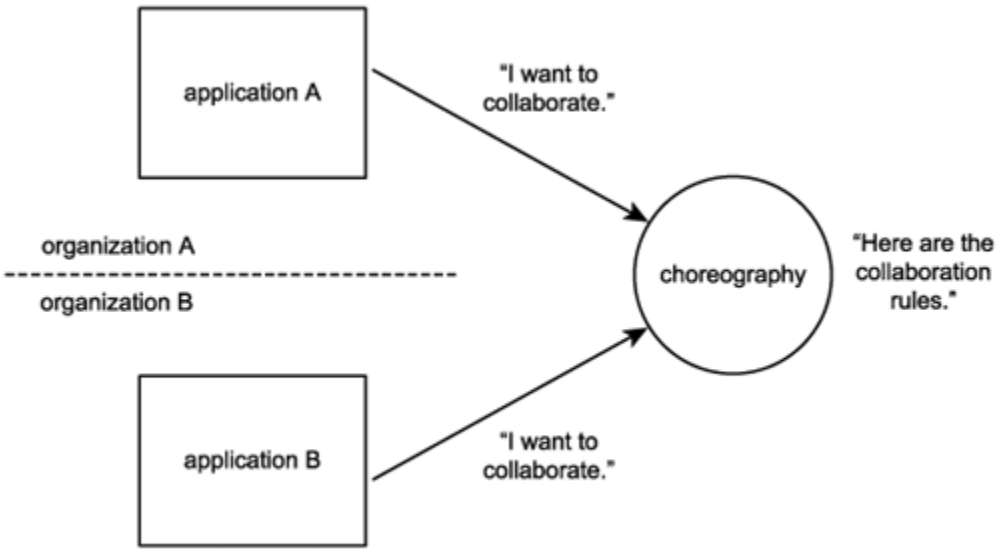
Choreography

In a perfect world, all organizations would agree on how internal processes should be structured, so that should they ever have to interoperate, they would already have their automation solutions in perfect alignment.

Though this vision has about a zero percent chance of ever becoming reality, the requirement for organizations to interoperate via services is becoming increasingly real and increasingly complex. This is especially true when interoperation requirements extend into the realm of collaboration, where multiple services from different organizations need to work together to achieve a common goal.

The Web Services Choreography Description Language (WS-CDL) is one of several specifications that attempts to organize information exchange between multiple organizations (or even multiple applications within organizations), with an emphasis on public collaboration It is the specification we've chosen here to represent the concept of choreography and also the specification from which many of the terms discussed in this section have been derived.

A choreography enables collaboration between its participants.



Collaboration

An important characteristic of choreographies is that they are intended for public message exchanges. The goal is to establish a kind of organized collaboration between services representing different service entities, only no one entity (organization) necessarily controls the collaboration logic. Choreographies therefore provide the potential for establishing universal interoperability patterns for common inter-organization business tasks.

Roles and participants

Within any given choreography, a Web service assumes one of a number of predefined roles. This establishes what the service does and what the service can do within the context of a particular business task. Roles can be bound to WSDL definitions, and those related are grouped accordingly, categorized as participants (services).

Relationships and channels

Every action that is mapped out within a choreography can be broken down into a series of message exchanges between two services. Each potential exchange between two roles in a choreography is therefore defined individually as a relationship. Every relationship consequently consists of exactly two roles.

Now that we've defined who can talk with each other, we require a means of establishing the nature of the conversation. Channels do exactly that by defining the characteristics of the message exchange between two specific roles.

Further, to facilitate more complex exchanges involving multiple participants, channel information can actually be passed around in a message. This allows one service to send another the information required for it to be communicated with by other services. This is a significant feature of the WS-CDL specification, as it fosters dynamic discovery and increases the number of potential participants within large-scale collaborative tasks.

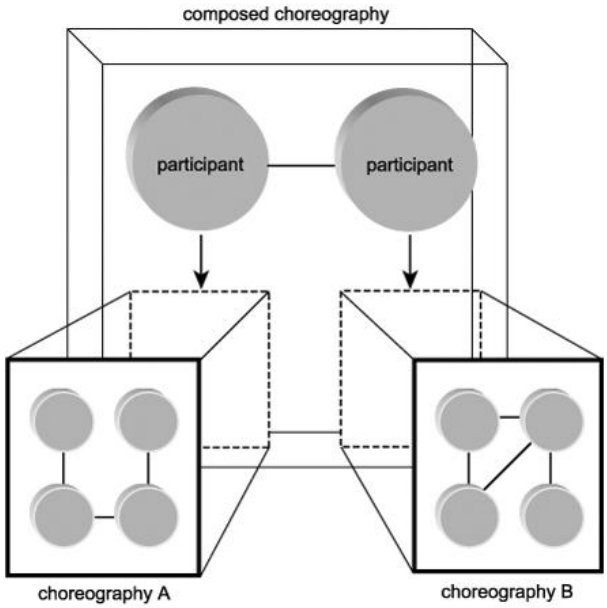
Interactions and work units

Finally, the actual logic behind a message exchange is encapsulated within an interaction. Interactions are the fundamental building blocks of choreographies because the completion of an interaction represents actual progress within a choreography. Related to interactions are work units. These impose rules and constraints that must be adhered to for an interaction to successfully complete.

Reusability, composability, and modularity

Each choreography can be designed in a reusable manner, allowing it to be applied to different business tasks comprised of the same fundamental actions. Further, using an import facility, a choreography can be assembled from independent modules. These modules can represent distinct sub-tasks and can be reused by numerous different parent choreographies

A choreography composed of two smaller choreographies.

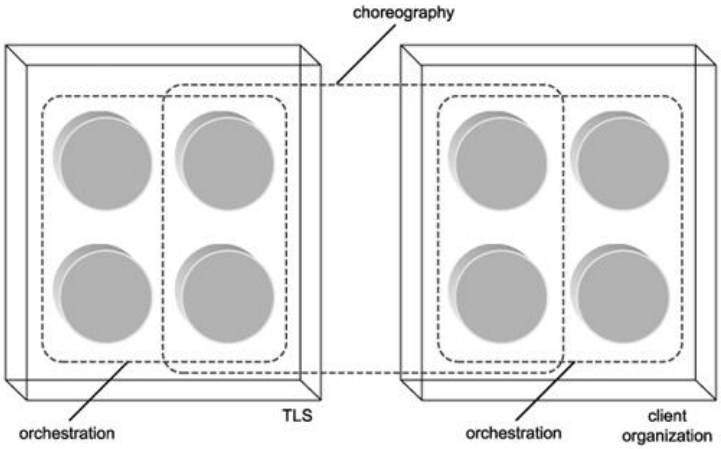


Finally, even though a choreography in effect composes a set of non-specific services to accomplish a task, choreographies themselves can be assembled into larger compositions.

Orchestrations and choreographies

While both represent complex message interchange patterns, there is a common distinction that separates the terms "orchestration" and "choreography." An orchestration expresses organization-specific business workflow. This means that an organization owns and controls the logic behind an orchestration, even if that logic involves interaction with external business partners. A choreography, on the other hand, is not necessarily owned by a single entity. It acts as a community interchange pattern used for collaborative purposes by services from different provider entities

A choreography enabling collaboration between two different orchestrations.



One can view an orchestration as a business-specific application of a choreography. This view is somewhat accurate, only it is muddled by the fact that some of the functionality provided by the corresponding specifications (WS-CDL and WS-BPEL) actually overlaps. This is a consequence of these specifications being developed in isolation and submitted to separate standards organizations (W3C and OASIS, respectively).

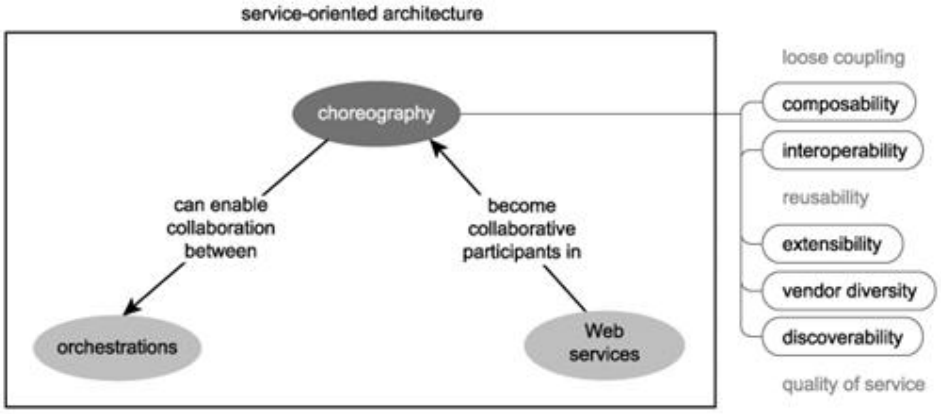
An orchestration is based on a model where the composition logic is executed and controlled in a centralized manner. A choreography typically assumes that there is no single owner of collaboration logic. However, one area of overlap between the current orchestration and choreography extensions is the fact that orchestrations can be designed to include multi-organization participants. An orchestration can therefore effectively establish cross-enterprise activities in a similar manner as a choreography. Again, though, a primary distinction is the fact that an orchestration is generally owned and operated by a single organization.

Choreography and SOA

The fundamental concept of exposing business logic through autonomous services can be applied to just about any implementation scope. Two services within a single organization, each exposing a simple function, can interact via a basic MEP to complete a simple task. Two services belonging to different organizations, each exposing functionality from entire enterprise business solutions, can interact via a basic choreography to complete a more complex task. Both scenarios involve two services, and both scenarios support SOA implementations.

Choreography therefore can assist in the realization of SOA across organization boundaries. While it natively supports composability, reusability, and extensibility, choreography also can increase organizational agility and discovery. Organizations are able to join into multiple online collaborations, which can dynamically extend or even alter related business processes that integrate with the choreographies. By being able to pass around channel information, participating services can make third-party organizations aware of other organizations with which they already have had contact.

Choreography relating to other parts of SOA.



WS-Policy language basics

The WS-Policy framework establishes a means of expressing service metadata beyond the WSDL definition. Specifically, it allows services to communicate rules and preferences in relation to security, processing, or message content. Policies can be applied to a variety of Web resources, positioning this specification as another fundamental part of the WS-* extensions discussed in this chapter

How WS-Policy relates to the other WS-* specifications discussed in this chapter.

The WS-Policy framework is comprised of the following three specifications:

- WS-Policy
- WS-PolicyAssertions
- WS-PolicyAttachments

These collectively provide the following elements covered in this section, which demonstrate how policies are formulated and attached to element or document-level subjects:

- `Policy` element
- `TextEncoding`, `Language`, `SpecVersion`, and `MessagePredicate` assertions
- `ExactlyOne` element
- `All` element
- `Usage` and `Preference` attributes

- `PolicyReference` element
- `PolicyURIs` attribute
- `PolicyAttachment` element

The `Policy` element and common policy assertions

The `Policy` element establishes the root construct used to contain the various policy assertions that comprise the policy. The WS-PolicyAssertions specification supplies the following set of common, predefined assertion elements:

- `TextEncoding` Dictates the use of a specific text encoding format.
- `Language` Expresses the requirement or preference for a particular language.
- `SpecVersion` Communicates the need for a specific version of a specification.
- `MessagePredicate` Indicates message processing rules expressed using XPath statements.

These elements represent assertions that can be used to structure basic policies around common requirements. Policy assertions also can be customized, and other WS-* specifications may provide supplemental assertions.

Each assertion can indicate whether its use is required or not via the value assigned to its `Usage` attribute. A value of "Required" indicates that its conditions must be met. Additionally, the use of the `Preference` attribute allows an assertion to communicate its importance in comparison to other assertions of the same type.

The `ExactlyOne` element

This construct surrounds multiple policy assertions and indicates that there is a choice between them, but that one must be chosen.

The `All` element

The `All` construct introduces a rule that states that all of the policy assertions within the construct must be met. This element can be combined with the `ExactlyOne` element, where collections of policy assertions can each be grouped into `All` constructs that are then further grouped into a parent `ExactlyOne` construct. This indicates that the policy is offering a choice of assertions groups but that the assertions in any one of the alternative `All` groups must be met.

The `Usage` attribute

As you've seen in the previous examples, a number of WS-Policy assertion elements contain a `Usage` attribute to indicate whether a given policy assertion is required. This attribute is a key part of the WS-Policy framework as its values form part of the overall policy rules.

Possible settings for the `Usage` attribute.

Attribute Value	Description
`Required`	The assertion requirements must be met, or an error will be generated.
`Optional`	The assertion requirements may be met, but an error will not be generated if they are not met.
`Rejected`	The assertion is unsupported.
`Observed`	The assertion applies to all policy subjects.
`Ignored`	The assertion will intentionally be ignored.

The `Preference` attribute

Policy assertions can be ranked in order of preference using this attribute. This is especially relevant if a service provider is flexible enough to provide multiple policy alternatives to potential service requestors.

The `Preference` attribute is assigned an integer value. The higher this value, the more preferred the assertion. When this attribute is not used, a default value of "0" is assigned to the policy assertion.

The `PolicyReference` element

So far we've only been discussing the creation of policy documents. However, we have not yet established how policies are associated with the subjects to which they apply. The `PolicyReference` element is one way to simply link an element with one or more policies. Each `PolicyReference` element contains a `URI` attribute that points to one policy document or a specific policy assertion within the document. (The `ID` attribute of the policy or grouping construct is referenced via the value displayed after the "#" symbol.)

If multiple `PolicyReference` elements are used within the same element, the policy documents are merged at runtime.

The `PolicyURIs` attribute

Alternatively, the `PolicyURIs` attribute also can be used to link to one or more policy documents. The attribute is added to an element and can be assigned multiple policy locations. As with `PolicyReference`, these policies are then merged at runtime.

The `PolicyAttachment` element

Another way of associating a policy with a subject is through the use of the `PolicyAttachment` construct. The approach taken here is that the child `AppliesTo` construct is positioned as the parent of the subject elements. The familiar `PolicyReference` element then follows the `AppliesTo` construct to identify the policy assertions that will be used.

Additional types of policy assertions

It is important to note that policy assertions can be utilized and customized beyond the conventional manner in which they are displayed in the preceding examples.

For example:

- Policy assertions can be incorporated into WSDL definitions through the use of a special set of policy subjects that target specific parts of the definition structure. A separate `UsingPolicy` element is provided for use as a WSDL extension.
- WS-ReliableMessaging defines and relies on WS-Policy assertions to enforce some of its delivery and acknowledgement rules.
- WS-Policy assertions can be created to communicate that a Web service is capable of participating in a business activity or an atomic transaction.
- A policy assertion can be designed to express a service's processing requirements in relation to other WS-* specifications.
- WS-Policy assertions commonly are utilized within the WS-Security framework to express security requirements.

WS-Security language basics

The WS-Security framework provides extensions that can be used to implement message-level security measures. These protect message contents during transport and during processing by service intermediaries. Additional extensions implement authentication and authorization control, which protect service providers from malicious requestors.

How WS-Security relates to the other WS-* specifications

The WS-Security framework is comprised of numerous specifications, many in different stages of acceptance and maturation. In this book we've concentrated on some of the more established ones, namely:

- WS-Security
- XML-Encryption
- XML-Signature

The `Security` element (WS-Security)

This construct represents the fundamental header block provided by WS-Security. The `Security` element can have a variety of child elements, ranging from XML-Encryption and XML-Signature constructs to the token elements provided by the WS-Security specification itself.

`Security` elements can be outfitted with `actor` attributes that correspond to SOAP actor roles This allows you to add multiple `Security` blocks to a SOAP message, each intended for a different recipient.

The `UsernameToken`, `Username`, and `Password` elements (WS-Security)

The `UsernameToken` element provides a construct that can be used to host token information for authentication and authorization purposes. Typical children of this construct are the `Username` and `Password` child elements, but custom elements also can be added.

The `BinarySecurityToken` element (WS-Security)

Tokens stored as binary data, such as certificates, can be represented in an encoded format within the `BinarySecurityToken` element.

The `SecurityTokenReference` element (WS-Security)

This element allows you to provide a pointer to a token that exists outside of the SOAP message document.

Composing `Security` element contents (WS-Security)

As previously mentioned, the WS-Security specification positions the `Security` element as a standardized container for header blocks originating from other security extensions. The following example illustrates this by showing how a SAML block is located within the `Security` construct. (As previously mentioned, single sign-on languages are beyond the scope of this book. The SAML-specific elements shown in this example therefore are not explained.)

The `EncryptedData` element (XML-Encryption)

This is the parent construct that hosts the encrypted portion of an XML document. If located at the root of an XML document, the entire document contents are encrypted.

The `CipherData`, `CipherValue`, and `CipherReference` elements (XML-Encryption)

The `CipherData` construct is required and must contain either a `CipherValue` element hosting the characters representing the encrypted text or a `CipherReference` element that provides a pointer to the encrypted values.

XML-Signature elements

A digital signature is a complex piece of information comprised of specific parts that each represent an aspect of the document being signed. Therefore, numerous elements can be involved when defining the construct that hosts the digital signature information.

XML-Signature elements

Element	Description
`CanonicalizationMethod`	This element identifies the type of "canonicalization algorithm" used to detect and represent subtle variances in the document content (such as the location of white space).

`DigestMethod`	Identifies the algorithm used to create the signature.
`DigestValue`	Contains a value that represents the document being signed. This value is generated by applying the `DigestMethod` algorithm to the XML document.
`KeyInfo`	This optional construct contains the public key information of the message sender.
`Signature`	The root element, housing all of the information for the digital signature.
`SignatureMethod`	The algorithm used to produce the digital signature. The digest and canonicalization algorithms are taken into account when creating the signature.
`SignatureValue`	The actual value of the digital signature.
`SignedInfo`	A construct that hosts elements with information relevant to the `SignatureValue` element, which resides outside of this construct.
`Reference`	Each document that is signed by the same digital signature is represented by a `Reference` construct that hosts digest and optional transformation details.