A

WS-BPEL 2.0 Syntax Reference

This appendix provides a syntax reference for the WS-BPEL Web Services Business Process Execution Language Version 2.0, OASIS Standard as defined in the specification dated April 11, 2007, available at the following URLs:

- `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`
- `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.doc`
- `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`

In the following sections, you will see the syntax for:

- BPEL activities and elements
- Default handlers
- BPEL functions
- Deadline and duration expressions
- Standard elements
- Standard attributes
- Default attribute values
- Standard faults
- Namespaces

Important BPEL activities and elements

The following section covers important activity and element tags associated with BPEL processing.

<assign>, <copy>, <from>,and <to>

The `<assign>` activity is used to:

- Copy data from one variable to another
- Construct and insert new data using expressions and literal values
- Copy partner link endpoint references

Syntax

```
<assign validate="yes|no"? standard-attributes>
  standard-elements
  (
  <copy keepSrcElementName="yes|no"?
    ignoreMissingFromData="yes|no"?>
    from-spec
    to-spec
  </copy>
  |
  <extensionAssignOperation>
    assign-element-of-other-namespace
  </extensionAssignOperation>
  )+
</assign>
```

The `from-spec` section can have the following forms:

```
<from variable="BPELVariableName" part="NCName"?>
  <query queryLanguage="anyURI"?>?
    queryContent
  </query>
</from>
<from partnerLink="NCName" endpointReference="myRole|partnerRole" />
<from variable="BPELVariableName" property="QName" />
<from expressionLanguage="anyURI"?>expression</from>
<from><literal>literal value</literal></from>
<from/>
```

The `to-spec` section can have the following forms:

```
<to variable="BPELVariableName" part="NCName"?>
  <query queryLanguage="anyURI"?>?
    queryContent
  </query>
</to>
```

```
<to partnerLink="NCName" />
<to variable="BPELVariableName" property="QName" />
<to expressionLanguage="anyURI"?>expression</to>
<to/>
```

Example

```
<assign name="PrepareInputForAAandDA">
  <copy>
    <from variable="TravelRequest" part="flightData"/>
    <to variable="FlightDetails" part="flightData"/>
  </copy>
  <copy>
    <from variable="EmployeeTravelStatusResponse"
      part="travelClass"/>
    <to variable="FlightDetails" part="travelClass"/>
  </copy>
</assign>
```

<catch>, <catchAll>

The `<catch>` activity is specified within fault handlers to specify faults that are to be caught and handled. The `<catchAll>` activity is used to catch all faults. Within a fault handler, at least one `<catch>` activity needs to be specified. The optional `<catchAll>` activity can also be specified.

The `<catch>` activity has four attributes that can be used to specify which fault to handle (at least one of them needs to be specified).

- `faultName`: Specifies the name of the fault to handle
- `faultVariable`: Specifies the variable type used for fault data
- `faultMessageType`: Specifies the message type used for fault data
- `faultElement`: Specifies the element used for fault data

Syntax

```
<faultHandlers>?
  <!-- Note: There must be at least one faultHandler -->
  <catch faultName="QName"?
    faultVariable="BPELVariableName"?
    ( faultMessageType="QName" | faultElement="QName" )? >*
        activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
</faultHandlers>
```

When used to define an inline fault handler for the `<invoke>` activity, the syntax is as follows:

```
<invoke partnerLink="NCName"
  portType="QName"?
  operation="NCName"
  inputVariable="BPELVariableName"?
  outputVariable="BPELVariableName"?
  standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="NCName" initiate="yes|join|no"?
      pattern="request|response|request-response"? />+
  </correlations>
  <catch faultName="QName"?
    faultVariable="BPELVariableName"?
    faultMessageType="QName"?
    faultElement="QName"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
  <compensationHandler>?
    activity
  </compensationHandler>
  <toParts>?
    <toPart part="NCName" fromVariable="BPELVariableName" />+
  </toParts>
  <fromParts>?
    <fromPart part="NCName" toVariable="BPELVariableName" />+
  </fromParts>
</invoke>
```

Example

```
<faultHandlers>
  <catch faultName="trv:TicketNotApproved" >
    <!-- Perform an activity -->
  </catch>

  <catch faultName="trv:TicketNotApproved"
    faultVariable="TravelFault" >
    <!-- Perform an activity -->
  </catch>

  <catch faultVariable="TravelFault" >
    <!-- Perform an activity -->
  </catch>

  <catchAll>
    <!-- Perform an activity -->
  </catchAll>
</faultHandlers>
```

<compensate>

To start compensation on all inner scopes that have already completed successfully, in default order, you use the `<compensate>` activity. We can use this activity from within a fault handler, another compensation handler, or a termination handler only.

Syntax

```
<compensate standard-attributes>
  standard-elements
</compensate>
```

Example

```
<compensate/>
```

<compensateScope>

To start compensation on a specified inner scope that has already completed successfully, you use the `<compensateScope>` activity. The `<compensateScope>` activity has an optional `target` attribute that is used to specify the scope to be compensated. We can use this activity from within a fault handler, another compensation handler, or a termination handler only.

Syntax

```
<compensateScope target="NCName" standard-attributes>
  standard-elements
</compensateScope>
```

Example

```
<compensateScope target="TicketConfirmationPayment" />
```

<compensationHandler>

Compensation handlers are used to define compensation activities. Compensation handlers gather all activities that have to be carried out to compensate another activity and can be defined for the whole process or scope, or can be inline for the `<invoke>` activity.

Syntax

```
<compensationHandler>
  activity
</compensationHandler>
```

Example

Compensation handler for a scope:

```
<scope name="TicketConfirmationPayment" >

  <compensationHandler>

    <invoke partnerLink="AmericanAirlines"
      portType="aln:TicketConfirmationPT"
      operation="CancelTicket"
      inputVariable="FlightDetails"
      outputVariable="Cancellation" />

    <invoke partnerLink="AmericanAirlines"
      portType="aln:TicketPaymentPT"
      operation="CancelPayment"
      inputVariable="PaymentDetails"
      outputVariable="PaymentCancellation" />

  </compensationHandler>
```

```
    <invoke partnerLink="AmericanAirlines"
      portType="aln:TicketConfirmationPT"
      operation="ConfirmTicket"
      inputVariable="FlightDetails"
      outputVariable="Confirmation" />

    <invoke partnerLink="AmericanAirlines"
      portType="aln:TicketPaymentPT"
      operation="PayTicket"
      inputVariable="PaymentDetails"
      outputVariable="PaymentConfirmation" />
  </scope>
```

Inline compensation handler for the `<invoke>` activity:

```
  <invoke name="TicketConfirmation"
    partnerLink="AmericanAirlines"
    portType="aln:TicketConfirmationPT"
    operation="ConfirmTicket"
    inputVariable="FlightDetails"
    outputVariable="Confirmation" >

    <compensationHandler>
      <invoke partnerLink="AmericanAirlines"
        portType="aln:TicketConfirmationPT"
        operation="CancelTicket"
        inputVariable="FlightDetails"
        outputVariable="Cancellation" >

    </compensationHandler>
  </invoke>
```

<correlations>, <correlation>

The `<correlation>` element is used to associate a correlation set (a collection of key data fields) with an activity. Correlation can be used within the `<receive>`, `<reply>`, `<invoke>`, `<onMessage>`, and `<onEvent>` activities.

Syntax

```
  <correlations>?
    <correlation set="NCName" initiate="yes|join|no"?
      pattern="request|response|request-response"? />+
  </correlations>
```

Example

```
<correlations>
  <correlation set="TicketOrder" initiate="yes"
    pattern="request-response" />
</correlations>
```

<correlationSets>, <correlationSet>

A correlation set is a set of properties shared by messages and used for correlation. It is used to associate a message with a business process instance. Each correlation set has a `name` attribute.

Syntax

```
<correlationSets>?
  <correlationSet name="NCName" properties="QName-list" />+
</correlationSets>
```

Example

```
<correlationSets>
  <correlationSet name="VehicleOrder"
    properties="tns:chassisNo tns:engineNo"/>
  <correlationSet name="TickerOrder"
    properties="aln:FlightNo"/>
</correlationSets>
```

<empty>

An activity that does nothing is defined by the `<empty>` tag.

Syntax

```
<empty standard-attributes>
  standard-elements
</empty>
```

Example

```
<empty/>
```

<eventHandlers>

Event handlers react to events that occur while the business process is executing. When these events occur, the corresponding event handlers are invoked. Event handlers can be specified for the whole process as well as for each scope.

Syntax

```
<eventHandlers>?
  <!-- Note: There must be at least one onEvent or onAlarm. -->
  <onEvent partnerLink="NCName"
    portType="QName"?
    operation="NCName"
    ( messageType="QName" | element="QName" )?
    variable="BPELVariableName"?
    messageExchange="NCName"?>*
    <correlations>?
      <correlation set="NCName" initiate="yes|join|no"? />+
    </correlations>
    <fromParts>?
      <fromPart part="NCName" toVariable="BPELVariableName" />+
    </fromParts>
    <scope ...>...</scope>
  </onEvent>
  <onAlarm>*
    <!-- Note: There must be at least one expression. -->
    (
    <for expressionLanguage="anyURI"?>duration-expr</for>
    |
    <until expressionLanguage="anyURI"?>deadline-expr</until>
    )?
    <repeatEvery expressionLanguage="anyURI"?>
      duration-expr
    </repeatEvery>?
    <scope ...>...</scope>
  </onAlarm>
</eventHandlers>
```

Example

```
<process name="Travel" ... >

  ...
  <eventHandlers>

    <onEvent partnerLink="client"
      portType="trv:TravelApprovalPT"
      operation="CancelTravelApproval"
      messageType="trv:TravelRequestMessage"
      variable="TravelRequest" >

        <scope>
            <exit/>
        </scope>
    </onEvent>

  </eventHandlers>
  ...
</process>
```

<exit>

The <exit> activity is used to immediately end a business process instance.

Syntax

```
<exit standard-attributes>
  standard-elements
</exit>
```

Example

```
<exit />
```

<extensionActivity>

The <extensionActivity> activity is used to extend BPEL by introducing a new activity type.

Syntax

```
<extensionActivity>
  <anyElementQName standard-attributes>
    standard-elements
  </anyElementQName>
</extensionActivity>
```

Example

```
<process name="Travel"...
  xmlns:human="http://uni-mb.si/bpel/humanInteractionExtensions">
   ...
   <extensionActivity>
      <!-- Non-standard activity nested within
           <extensionActivity> element -->
      <human:humanInteraction type="human:simpleApproval">
         <human:participantResolution role="ApprovalManager" />
         ...
      </human:humanInteraction>
   </extensionActivity>
   ...
</process>
```

<faultHandlers>, <faultHandler>

Fault handlers are used to react to faults that occur while the business process activities are executing. They can be specified for the global process or each scope, or inline for `<invoke>` activities. Multiple `<catch>` activities can be specified within the fault handler for specific faults. You need to specify at least one `<catch>` activity. You can optionally specify the `<catchAll>` activity. The syntax and functionality of the `<catch>` activity is the same as described in the earlier section on `<catch>` and `<catchAll>`.

Syntax

```
<faultHandlers>?
  <!-- Note: There must be at least one faultHandler -->
  <catch faultName="QName"?
    faultVariable="BPELVariableName"?
    ( faultMessageType="QName" | faultElement="QName" )? >*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
</faultHandlers>
```

An inline fault handler for the <invoke> activity is specified as shown below:

```
<invoke partnerLink="NCName"
  portType="QName"?
  operation="NCName"
  inputVariable="BPELVariableName"?
  outputVariable="BPELVariableName"?
  standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="NCName" initiate="yes|join|no"?
      pattern="request|response|request-response"? />+
  </correlations>
  <catch faultName="QName"?
    faultVariable="BPELVariableName"?
    faultMessageType="QName"?
    faultElement="QName"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
  <compensationHandler>?
    activity
  </compensationHandler>
  <toParts>?
    <toPart part="NCName" fromVariable="BPELVariableName" />+
  </toParts>
  <fromParts>?
    <fromPart part="NCName" toVariable="BPELVariableName" />+
  </fromParts>
</invoke>
```

Example

```
<faultHandlers>
  <catch faultName="trv:TicketNotApproved" >
    <!-- Perform an activity -->
  </catch>

  <catch faultName="trv:TicketNotApproved"
    faultVariable="TravelFault" >
    <!-- Perform an activity -->
  </catch>

  <catch faultVariable="TravelFault" >
    <!-- Perform an activity -->
  </catch>

  <catchAll>
    <!-- Perform an activity -->
  </catchAll>
</faultHandlers>
```

<flow>

The `<flow>` activity provides concurrent execution of enclosed activities and their synchronization.

Syntax

```
<flow standard-attributes>
  standard-elements
  <links>?
    <link name="NCName" />+
  </links>
  activity+
</flow>
```

Example

```
<!-- Make a concurrent invocation to AA in DA -->
<flow name="InvokeAAandDA">

  <sequence>
    <!-- Async invoke of the AA web service and wait for the
      callback-->

    <invoke name="InvokeAA"
      partnerLink="AmericanAirlines"
      portType="aln:FlightAvailabilityPT"
      operation="FlightAvailability"
      inputVariable="FlightDetails" />

    <receive name="ReceiveCallbackFromAA"
      partnerLink="AmericanAirlines"
      portType="aln:FlightCallbackPT"
      operation="FlightTicketCallback"
      variable="FlightResponseAA" />

  </sequence>

  <sequence>
    <!-- Async invoke of the DA web service and wait for the
      callback-->

    <invoke name="InvokeDA"
      partnerLink="DeltaAirlines"
      portType="aln:FlightAvailabilityPT"
      operation="FlightAvailability"
      inputVariable="FlightDetails" />

    <receive name="ReceiveCallbackFromDA"
       partnerLink="DeltaAirlines"
       portType="aln:FlightCallbackPT"
       operation="FlightTicketCallback"
       variable="FlightResponseDA" />

  </sequence>

</flow>
```

<forEach>

A <forEach> activity iterates its child scope activity exactly N+1 times where N equals the <finalCounterValue> minus the <startCounterValue> value. The <forEach> branches can execute in parallel, if parallel="yes". A <completionCondition> may be used within the <forEach> to allow the <forEach> activity to complete without executing or finishing all the branches specified.

Syntax

```
<forEach counterName="BPELVariableName" parallel="yes|no"

  standard-attributes>
  standard-elements
  <startCounterValue expressionLanguage="anyURI"?>
    unsigned-integer-expression
  </startCounterValue>
  <finalCounterValue expressionLanguage="anyURI"?>
    unsigned-integer-expression
  </finalCounterValue>
  <completionCondition>?
    <branches expressionLanguage="anyURI"?
      successfulBranchesOnly="yes|no"?>?
        unsigned-integer-expression
    </branches>
  </completionCondition>
  <scope ...>...</scope>
</forEach>
```

Example

```
<forEach counterName="Counter" parallel="no">

  <startCounterValue>number(1)</startCounterValue>
  <finalCounterValue>$NoOfPassengers</finalCounterValue>

  <scope>
    <sequence>

    <!-- Construct the FlightDetails variable with passenger data -->
    ...

    <!-- Invoke the web service -->
        <invoke partnerLink="AmericanAirlines"
```

```
            portType="aln:FlightAvailabilityPT"
            operation="FlightAvailability"
            inputVariable="FlightDetails" />

          <receive partnerLink="AmericanAirlines"
            portType="trv:FlightCallbackPT"
            operation="FlightTicketCallback"
            variable="FlightResponseAA" />

      ...
      <!-- Process the results ... -->
      ...

      </sequence>
    </scope>

  </forEach>
```

<fromParts>, <fromPart>

The <fromParts> activity is used to specify how to store the different message parts to BPEL variables. We can specify <fromParts> in the following BPEL activities that receive messages:

- <receive>
- <onMessage> (within <pick>)
- <onEvent> (within <eventHandlers>)
- <invoke> (for receiving the response in request/response operations).

Syntax

```
<fromParts>?
  <fromPart part="NCName" toVariable="BPELVariableName" />+
</fromParts>
```

Example

```
<receive partnerLink="AmericanAirlines"
  portType="aln:FlightCallbackPT"
  operation="FlightTicketCallback" >
  <correlations>
    <correlation set="TicketOrder"
      initiate="yes" />
  </correlations>
```

```
    <fromParts
      <fromPart part="confirmationData" toVariable="TravelResponse" />
    </fromParts>

  </receive>
```

<if>, <elseif>, and <else>

To express conditional behavior, the `<if>` activity is used. It consists of one or more conditional branches defined by `<elseif>` elements, followed by an optional `<else>` element.

Syntax

```
<if standard-attributes>
  standard-elements
  <condition expressionLanguage="anyURI"?>bool-expr</condition>
  activity
  <elseif>*
    <condition expressionLanguage="anyURI"?>bool-expr</condition>
    activity
  </elseif>
  <else>?
    activity
  </else>
</if>
```

Example

```
<!-- Select the best offer and construct the TravelResponse -->
<if>
  <condition>
    $FlightResponseAA.confirmationData/aln:Price &lt;=
    $FlightResponseDA.confirmationData/aln:Price
  </condition>
  <!-- Select American Airlines -->
  <assign>
    <copy>
       <from variable="FlightResponseAA" />
       <to variable="TravelResponse" />
    </copy>
  </assign>
```

```
    <else>
      <!-- Select Delta Airlines -->
      <assign>
          <copy>
              <from variable="FlightResponseDA" />
              <to variable="TravelResponse" />
          </copy>
      </assign>
    </else>
  </if>
```

<import>

The `<import>` activity is used within a BPEL process to declare a dependency on external XML Schema or WSDL definitions. Any number of `<import>` elements may appear as children of the `<process>` element.

Syntax

```
<import namespace="anyURI"?
  location="anyURI"?
  importType="anyURI" />*
```

Example

```
<process name="Travel" ...
         xmlns:ame="http://packtpub.com/service/airline/" />
   <import importType="http://schemas.xmlsoap.org/wsdl/"
           location="http://localhost/services/AmericanAirlines.wsdl"
           namespace="http://packtpub.com/service/airline/" />
   ...
</process>
```

<invoke>

The `<invoke>` activity is used to invoke the web service operations provided by partners.

Syntax

```
<invoke partnerLink="NCName"
  portType="QName"?
  operation="NCName"
  inputVariable="BPELVariableName"?
  outputVariable="BPELVariableName"?
  standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="NCName" initiate="yes|join|no"?
      pattern="request|response|request-response"? />+
  </correlations>
  <catch faultName="QName"?
    faultVariable="BPELVariableName"?
    faultMessageType="QName"?
    faultElement="QName"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
  <compensationHandler>?
    activity
  </compensationHandler>
  <toParts>?
    <toPart part="NCName" fromVariable="BPELVariableName" />+
  </toParts>
  <fromParts>?
    <fromPart part="NCName" toVariable="BPELVariableName" />+
  </fromParts>
</invoke>
```

Example

```
<!-- Synchronously invoke the Employee Travel Status Web Service -->
<invoke name="EmployeeTravelStatusSyncInv"
  partnerLink="employeeTravelStatus"
  portType="emp:EmployeeTravelStatusPT"
  operation="EmployeeTravelStatus"
  inputVariable="EmployeeTravelStatusRequest"
  outputVariable="EmployeeTravelStatusResponse" />

<!-- Async invoke of the AA web service -->
<invoke name="AmericanAirlinesAsyncInv"
  partnerLink="AmericanAirlines"
  portType="aln:FlightAvailabilityPT"
  operation="FlightAvailability"
  inputVariable="FlightDetails" />
```

<links>, <link>

Synchronization dependencies in concurrent flows are specified using <links>.

Syntax

```
<links>?
  <link name="NCName" />+
</links>
```

Example

```
<links>
  <link name="TravelStatusToTicketRequest" />
  <link name="TicketRequestToTicketConfirmation" />
</links>
```

<messageExchanges>, <messageExchange>

The <messageExchange> is used to associate an outbound message activity (<reply>) activity with an inbound message activity (<receive>, <onMessage>, or <onEvent>).

Syntax

```
<messageExchanges>?
  <messageExchange name="NCName" />+
</messageExchanges>
```

Example

```
<messageExchanges>
  <messageExchange name="TravelExchange" />
</messageExchanges>
...
<receive partnerLink="client" portType="trv:TravelApprovalPT"
  operation="TravelApproval"
  variable="TravelRequest"

  messageExchange="TravelExchange" />
...
<reply partnerLink="client"
  portType="trv:TravelApprovalPT"
  operation="TravelApproval"
  variable="TravelResponse"
  messageExchange="TravelExchange" />
```

<onAlarm>

This activity is used in the <pick> and <eventHandlers> activities to specify the occurrence of alarm events.

Syntax in <pick>

```
<onAlarm>*
  (
  <for expressionLanguage="anyURI"?>duration-expr</for>
  |
  <until expressionLanguage="anyURI"?>deadline-expr</until>
  )
  activity
</onAlarm>
```

Syntax in <eventHandlers>

```
<onAlarm>*
  <!-- Note: There must be at least one expression. -->
  (
  <for expressionLanguage="anyURI"?>duration-expr</for>
  |
  <until expressionLanguage="anyURI"?>deadline-expr</until>
  )?
  <repeatEvery expressionLanguage="anyURI"?>
    duration-expr
  </repeatEvery>?
  <scope ...>...</scope>
</onAlarm>
```

Example

```
<onAlarm>
  <for>'PT15M'</for>
  <!-- Perform an activity or a set of activities enclosed by
    <sequence>, <flow>, etc. or throw a fault -->
</onAlarm>
```

<onEvent>

This is used in <eventHandlers> activities to specify the occurrence of message events.

Syntax

```
<onEvent partnerLink="NCName"
  portType="QName"?
  operation="NCName"
  ( messageType="QName" | element="QName" )?
  variable="BPELVariableName"?
  messageExchange="NCName"?>*
  <correlations>?
    <correlation set="NCName" initiate="yes|join|no"? />+
  </correlations>
  <fromParts>?
    <fromPart part="NCName" toVariable="BPELVariableName" />+
  </fromParts>
  <scope ...>...</scope>
</onEvent>
```

Example

```
<onEvent partnerLink="client"
  portType="trv:TravelApprovalPT"
  operation="CancelTravelApproval"
  messageType="trv:TravelRequestMessage"
  variable="TravelRequest" >

  <scope>
    <exit/>
  </scope>
</onEvent>
```

<onMessage>

This is used in `<pick>` activity to specify the occurrence of message events.

Syntax

```
<onMessage partnerLink="NCName"
  portType="QName"?
  operation="NCName"
  variable="BPELVariableName"?
  messageExchange="NCName"?>+
  <correlations>?
    <correlation set="NCName" initiate="yes|join|no"? />+
  </correlations>
  <fromParts>?
    <fromPart part="NCName" toVariable="BPELVariableName" />+
  </fromParts>
  activity
</onMessage>
```

Example

```
<onMessage partnerLink="AmericanAirlines"
  portType="aln:FlightCallbackPT"
  operation="FlightNotAvailable"
  variable="FlightFaultAA">
  <throw faultName="trv:FlightNotAvailable"
    faultVariable="FlightFaultAA"/>
</onMessage>
```

<partnerLinks>, <partnerLink>

A business process interacts with services that are modeled as partner links. Each partner link is characterized by a `<partnerLinkType>`. More than one partner link can be characterized by the same `<partnerLinkType>`. See the next section for more on `<partnerLinkType>`.

Syntax

```
<partnerLinks>?
  <!-- Note: At least one role must be specified. -->
  <partnerLink name="NCName"
    partnerLinkType="QName"
    myRole="NCName"?
    partnerRole="NCName"?
    initializePartnerRole="yes|no"?>+
  </partnerLink>
</partnerLinks>
```

Example

```
<partnerLinks>
  <partnerLink name="insurance"
    partnerLinkType="tns:insuranceLT"
    myRole="insuranceRequester"
    partnerRole="insuranceService"/>
</partnerLinks>
```

<partnerLinkType>, <role>

A partner link type characterizes the relationship between two services. It defines roles for each of the services in the conversation between them and specifies the port type provided by each service to receive messages. Partner link types and roles are specified in the WSDL.

Syntax

```
<wsdl:definitions name="NCName" targetNamespace="anyURI" ...>
  ...
  <plnk:partnerLinkType name="NCName">
    <plnk:role name="NCName" portType="QName" />
    <plnk:role name="NCName" portType="QName" />?
  </plnk:partnerLinkType>
  ...
</wsdl:definitions>
```

Example

```
<plnk:partnerLinkType name="insuranceLT">
  <plnk:role name="insuranceService"
    portType ="ins:ComputeInsurancePremiumPT"/>
  <plnk:role name="insuranceRequester"
    portType ="ins:ComputeInsurancePremiumCallbackPT"/>
</plnk:partnerLinkType>
```

<pick>

The <pick> activity is used to wait for the occurrence of one of a set of events and then perform an activity associated with the event.

Syntax

```
<pick createInstance="yes|no"? standard-attributes>
  standard-elements
  <onMessage partnerLink="NCName"
    portType="QName"?
    operation="NCName"
    variable="BPELVariableName"?
    messageExchange="NCName"?>+
    <correlations>?
        <correlation set="NCName" initiate="yes|join|no"? />+
    </correlations>
    <fromParts>?
        <fromPart part="NCName" toVariable="BPELVariableName" />+
    </fromParts>
    activity
  </onMessage>
  <onAlarm>*
      (
      <for expressionLanguage="anyURI"?>duration-expr</for>
```

```
        |
        <until expressionLanguage="anyURI"?>deadline-expr</until>
        )
        activity
    </onAlarm>
</pick>
```

Example

```
<pick>
  <onMessage partnerLink="AmericanAirlines"
    portType="aln:FlightCallbackPT"
    operation="FlightTicketCallback"
    variable="FlightResponseAA">
    <empty/>
    <!-- Continue with the rest of the process -->
  </onMessage>

  <onMessage partnerLink="AmericanAirlines"
    portType="aln:FlightCallbackPT"
    operation="FlightNotAvailable"
    variable="FlightFaultAA">
    <throw faultName="trv:FlightNotAvailable"
      faultVariable="FlightFaultAA"/>
  </onMessage>

  <onMessage partnerLink="AmericanAirlines"
    portType="aln:FlightCallbackPT"
    operation="TicketNotAvailable"
    variable="FlightFaultAA">
    <throw faultName="trv:TicketNotAvailable"
      faultVariable="FlightFaultAA"/>
  </onMessage>

  <onAlarm>
    <for>'PT30M'</for>
    <throw faultName="trv:CallbackTimeout" />
  </onAlarm>
</pick>
```

<process>

This is the root element of each BPEL process definition.

Syntax

```
<process name="NCName" targetNamespace="anyURI"
  queryLanguage="anyURI"?
  expressionLanguage="anyURI"?
  suppressJoinFailure="yes|no"?
  exitOnStandardFault="yes|no"?
  xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

  <extensions>?
    <extension namespace="anyURI" mustUnderstand="yes|no" />+
  </extensions>

  <import namespace="anyURI"?
    location="anyURI"?
    importType="anyURI" />*

  <partnerLinks>?
    <!-- Note: At least one role must be specified. -->
    <partnerLink name="NCName"
      partnerLinkType="QName"
      myRole="NCName"?
      partnerRole="NCName"?
      initializePartnerRole="yes|no"?>+
    </partnerLink>
  </partnerLinks>

  <messageExchanges>?
    <messageExchange name="NCName" />+
  </messageExchanges>

  <variables>?
    <variable name="BPELVariableName"
      messageType="QName"?
      type="QName"?
      element="QName"?>+
      from-spec?
    </variable>
  </variables>

  <correlationSets>?
    <correlationSet name="NCName" properties="QName-list" />+
  </correlationSets>

  <faultHandlers>?
```

```
    <!-- Note: There must be at least one faultHandler -->
    <catch faultName="QName"?
      faultVariable="BPELVariableName"?
      ( faultMessageType="QName" | faultElement="QName" )? >*
      activity
    </catch>
    <catchAll>?
        activity
    </catchAll>
  </faultHandlers>

  <eventHandlers>?
    <!-- Note: There must be at least one onEvent or onAlarm. -->
    <onEvent partnerLink="NCName"
      portType="QName"?
      operation="NCName"
      ( messageType="QName" | element="QName" )?
      variable="BPELVariableName"?
      messageExchange="NCName"?>*
        <correlations>?
            <correlation set="NCName" initiate="yes|join|no"? />+
        </correlations>
        <fromParts>?
            <fromPart part="NCName" toVariable="BPELVariableName" />+
        </fromParts>
        <scope ...>...</scope>
    </onEvent>
    <onAlarm>*
        <!-- Note: There must be at least one expression. -->
        (
        <for expressionLanguage="anyURI"?>duration-expr</for>
        |
        <until expressionLanguage="anyURI"?>deadline-expr</until>
        )?
        <repeatEvery expressionLanguage="anyURI"?>
            duration-expr
        </repeatEvery>?
        <scope ...>...</scope>
    </onAlarm>
  </eventHandlers>
  activity
</process>
```

Example

Examples are shown in chapters 2 and 3.

<property>

Properties are used to create globally unique names and associate them with data types (XML Schema types). Properties have greater significance than the types themselves. Properties are defined in WSDL.

Syntax

```
<wsdl:definitions name="NCName">
  <vprop:property name="NCName" type="QName"? element="QName"? />
  ...
</wsdl:definitions>
```

Example

```
<vprop:property name="FlightNo" type="xs:string" />
```

<propertyAlias>

Property aliases are used to map global properties to fields in specific message parts. Property aliases are defined in the WSDL.

Syntax

```
<wsdl:definitions name="NCName" ...>

  <vprop:propertyAlias propertyName="QName"
    messageType="QName"?
    part="NCName"?
    type="QName"?
    element="QName"?>
    <vprop:query queryLanguage="anyURI"?>?
        queryContent
    </vprop:query>
  </vprop:propertyAlias>
  ...

</wsdl:definitions>
```

Example

```
<vprop:propertyAlias propertyName="tns:FlightNo"
  messageType="tns:TravelResponseMessage"
  part="confirmationData">

  <vprop:query>
    /confirmationData/FlightNo
  </vprop:query>
</vprop:propertyAlias>
```

<receive>

A `<receive>` activity is used to receive requests in a BPEL business process to provide services to its partners.

Syntax

```
<receive partnerLink="NCName"
  portType="QName"?
  operation="NCName"
  variable="BPELVariableName"?
  createInstance="yes|no"?
  messageExchange="NCName"?
  standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="NCName" initiate="yes|join|no"? />+
  </correlations>
  <fromParts>?
    <fromPart part="NCName" toVariable="BPELVariableName" />+
  </fromParts>
</receive>
```

Example

```
<!-- Receive the initial request for business travel from client -->
<receive partnerLink="client"
  portType="trv:TravelApprovalPT"
  operation="TravelApproval"
  variable="TravelRequest" />
```

<repeatUntil>

A <repeatUntil> activity is used to define an iterative activity. The iterative activity is repeated until the specified <condition> becomes true. The <condition> is tested after the child activity completes. The <repeatUntil> activity is used to execute the child activity at least once.

Syntax

```
<repeatUntil standard-attributes>
  standard-elements
  activity
  <condition expressionLanguage="anyURI"?>bool-expr</condition>
</repeatUntil>
```

Example

```
<repeatUntil>

  <sequence>

  <!-- Construct the FlightDetails variable with passenger data -->
  ...

  <!-- Invoke the web service -->
    <invoke partnerLink="AmericanAirlines"
      portType="aln:FlightAvailabilityPT"
      operation="FlightAvailability"
      inputVariable="FlightDetails" />

    <receive partnerLink="AmericanAirlines"
      portType="trv:FlightCallbackPT"
      operation="FlightTicketCallback"
      variable="FlightResponseAA" />

    ...
    <!-- Process the results ... -->
    ...

    <!-- Increment the counter -->
    <assign>
        <copy>
            <from>$Counter + 1</from>
            <to variable="Counter"/>
        </copy>
    </assign>
  </sequence>

  <condition>$Counter &gt;= $NoOfPassengers</condition>
</repeatUntil>
```

<reply>

A <reply> activity is used to send a response to a request previously accepted through a <receive> activity. Responses are used for synchronous request/reply interactions. An asynchronous response is always sent by invoking the corresponding one-way operation on the partner link.

Syntax

```
<reply partnerLink="NCName"
   portType="QName"?
   operation="NCName"
   variable="BPELVariableName"?
   faultName="QName"?
   messageExchange="NCName"?
   standard-attributes>
   standard-elements
   <correlations>?
      <correlation set="NCName" initiate="yes|join|no"? />+
   </correlations>
   <toParts>?
      <toPart part="NCName" fromVariable="BPELVariableName" />+
   </toParts>
</reply>
```

Example

```
<!-- Send a response to the client -->
<reply partnerLink="client"
  portType="trv:TravelApprovalPT"
  operation="TravelApproval"
  variable="TravelResponse"/>
```

<rethrow>

The <rethrow> activity is used to rethrow the fault that was originally caught by the immediately enclosing fault handler.

Syntax

```
<rethrow standard-attributes>
  standard-elements
</rethrow>
```

Example

```
<rethrow />
```

<scope>

Scopes define behavior contexts for activities. They provide local partner links, message exchanges, variables, correlation sets, fault handlers, compensation handlers, termination handlers, and event handlers for activities.

Syntax

```
<scope isolated="yes|no"? exitOnStandardFault="yes|no"?
   standard-attributes>
   standard-elements
   <partnerLinks>?
      ... see above under <process> for syntax ...
   </partnerLinks>
   <messageExchanges>?
      ... see above under <process> for syntax ...
   </messageExchanges>
   <variables>?
      ... see above under <process> for syntax ...
   </variables>
   <correlationSets>?
      ... see above under <process> for syntax ...
   </correlationSets>
   <faultHandlers>?
      ... see above under <process> for syntax ...
   </faultHandlers>
   <compensationHandler>?
      ...
   </compensationHandler>
   <terminationHandler>?
      ...
   </terminationHandler>
   <eventHandlers>?
      ... see above under <process> for syntax ...
   </eventHandlers>
   activity
</scope>
```

Example

```
<scope>

  <faultHandlers>
    <catch faultName="emp:WrongEmployeeName" >
        <!-- Perform an activity -->
    </catch>
    <catch faultName="emp:TravelNotAllowed"
      faultVariable="Description" >
        <!-- Perform an activity -->
    </catch>
    <catchAll>
        <!-- Perform an activity -->
    </catchAll>
  </faultHandlers>

  <invoke partnerLink="employeeTravelStatus"
    portType="emp:EmployeeTravelStatusPT"
    operation="EmployeeTravelStatus"
    inputVariable="EmployeeTravelStatusRequest"
    outputVariable="EmployeeTravelStatusResponse" />
</scope>
```

<sequence>

A <sequence> activity is used to define activities that need to be performed in a sequential order.

Syntax

```
<sequence standard-attributes>
  standard-elements
  activity+
</sequence>
```

Example

Examples are covered in chapters 2 and 3.

<sources>, <source>

<source> elements are used to declare that an activity is the source of one or more links. This is a standard element.

Syntax

```
<sources>?
  <source linkName="NCName">+
    <transitionCondition expressionLanguage="anyURI"?>?
        bool-expr
    </transitionCondition>
  </source>
</sources>
```

Example

```
<invoke name="EmployeeTravelStatusSyncInv"
  partnerLink="employeeTravelStatus"
  portType="emp:EmployeeTravelStatusPT"
  operation="EmployeeTravelStatus"
  inputVariable="EmployeeTravelStatusRequest"
  outputVariable="EmployeeTravelStatusResponse" >

  <targets>
    <target linkName="EmployeeInputToEmployeeTravelStatusSyncInv" />
  </targets>
  <sources>
    <source linkName="EmployeeTravelStatusSyncInvToAirlinesInput" />
  </sources>

</invoke>
```

<targets>, <target>

The <target> element is used to declare that an activity is the target of one or more links. This is a standard element.

Syntax

```
<targets>?
  <joinCondition expressionLanguage="anyURI"?>?
    bool-expr
  </joinCondition>
  <target linkName="NCName" />+
</targets>
```

Example

```
<!-- Select the best offer and construct the TravelResponse -->
<if>

  <targets>
    <target linkName="AmericanAirlinesCallbackToBestOfferSelect" />
    <target linkName="DeltaAirlinesCallbackToBestOfferSelect" />
  </targets>

  <condition>
    $FlightResponseAA.confirmationData/aln:Price &lt;=
    $FlightResponseDA.confirmationData/aln:Price
  </condition>            <!-- Select American Airlines -->
  <assign>
    <copy>
        <from variable="FlightResponseAA" />
        <to variable="TravelResponse" />
    </copy>
  </assign>

  <else>
    <!-- Select Delta Airlines -->
    <assign>
        <copy>
            <from variable="FlightResponseDA" />
            <to variable="TravelResponse" />
        </copy>
    </assign>
  </otherwise>
</if>
```

<terminationHandler>

The `<terminationHandler>` element provide the ability for scopes to control the semantics of forced termination to some degree. It can be declared within `<scope>`.

Syntax

```
<terminationHandler>
  activity
</terminationHandler>
```

Example

```
<scope>
  <terminationHandler>
    <!-- clean up resources in case of termination -->
    ...
  </terminationHandler>
  <sequence>
    <!-- do some work -->
    ...
  </sequence>
</scope>
```

<throw>

The <throw> activity is used to generate a fault from inside the BPEL process.

Syntax

```
<throw faultName="QName"
  faultVariable="BPELVariableName"?
  standard-attributes>
  standard-elements
</throw>
```

Example

```
<throw faultName="WrongEmployeeName" />

<throw faultName="trv:TicketNotApproved"
  faultVariable="TravelFault" />
```

<toParts>, <toPart>

The <toParts> specifies which BPEL variables will map to which message part. The <toParts> activity is used to specify the mapping of outbound messages. We can specify <toParts> in the following BPEL activities—<reply>, <invoke>.

Syntax

```
<toParts>?
  <toPart part="NCName" fromVariable="BPELVariableName" />+
</toParts>
```

Example

```
<invoke partnerLink="AmericanAirlines"
  portType="aln:FlightAvailabilityPT"
  operation="FlightAvailability"
  inputVariable="FlightDetails" >
  <toParts>
    <toPart part="flightData" fromVariable="FlightDataDetails" />
    <toPart part="travelClass" fromVariable="FlightTravelClass" />
  </toParts>
</invoke>
```

<validate>

The <validate> activity is used to ensure that values of variables are valid against their associated XML data definition.

Syntax

```
<validate variables="BPELVariableNames" standard-attributes>
  standard-elements
</validate>
```

Example

```
<validate variables="InsuredPersonRequest InsuranceRequest
  PartialInsuranceDescription " />
```

<variables>, <variable>

Variables are used to hold messages that constitute the state of a business process. The variable may be of the WSDL message type, an XML Schema simple type, or an XML Schema element.

Syntax

```
<variables>
  <variable name="BPELVariableName"
    messageType="QName"?
    type="QName"?
    element="QName"?>+
    from-spec?
  </variable>
</variables>
```

Example

```
<variables>
  <variable name="InsuranceRequest"
    messageType="ins:InsuranceRequestMessage"/>
  <variable name="PartialInsuranceDescription"
    element="ins:InsuranceDescription"/>
  <variable name="lastName"
    type="xs:string"/>
</variables>
```

<wait>

A `<wait>` activity is used to specify a delay for a certain period of time or until a certain deadline is reached.

Syntax

```
<wait standard-attributes>
  standard-elements
  (
  <for expressionLanguage="anyURI"?>duration-expr</for>
  |
  <until expressionLanguage="anyURI"?>deadline-expr</until>
  )
</wait>
```

Examples

```
<wait>
  <until>'2004-03-18T21:00:00+01:00'</until>
</wait>
<wait>
  <until>'18:05:30Z'</until>
</wait>
<wait>
  <for>'PT4H10M'</for>
</wait>
<wait>
  <for>'P1M3DT4H10M'</for>
</wait>
<wait>
  <for>'P1Y11M14DT4H10M30S'</for>
</wait>
```

<while>

A <while> activity is used to define an iterative activity. The iterative activity is repeated as long as the specified <condition> is true.

Syntax

```
<while standard-attributes>
  standard-elements
  <condition expressionLanguage="anyURI"?>bool-expr</condition>
  activity
</while>
```

Example

```
<while>
  <condition>$Counter &lt; $NoOfPassengers</condition>
  <sequence>
  <!-- Construct the FlightDetails variable with passenger data -->
  ...
  <!-- Invoke the web service -->
    <invoke partnerLink="AmericanAirlines"
      portType="aln:FlightAvailabilityPT"
      operation="FlightAvailability"
      inputVariable="FlightDetails" />
    <receive partnerLink="AmericanAirlines"
      portType="trv:FlightCallbackPT"
      operation="FlightTicketCallback"
      variable="FlightResponseAA" />
    ...
    <!-- Process the results ... -->
    ...
    <!-- Increment the counter -->
    <assign>
        <copy>
            <from>$Counter + 1</from>
            <to variable="Counter"/>
        </copy>
    </assign>
  </sequence>
</while>
```

Default handlers

If fault handlers, compensation handlers, or termination handlers are not defined for a given `<scope>`, the BPEL engine automatically generates default handlers, as described in the following sections.

Default fault handler

If a `<catchAll>` fault handler for any fault is not defined for any given `<scope>`, the BPEL engine implicitly creates a default fault handler. The default fault handler will compensate all inner scopes and rethrow the fault to the upper scope. The default implicit fault handler looks as follows:

```
<catchAll>
   <sequence>
      <compensate />
      <rethrow />
   </sequence>
</catchAll>
```

Default compensation handler

If a `<compensationHandler>` is not defined for any given `<scope>`, the BPEL engine implicitly creates a default compensation handler, which compensates all inner scopes. The default compensation handler looks as follows:

```
<compensationHandler>
  <compensate />
</compensationHandler>
```

Default termination handler

If a custom `<terminationHandler>` for the scope is not present, BPEL engine will generate a default termination handler. The default termination handler has the following syntax:

```
<terminationHandler>
  <compensate />
</terminationHandler>
```

BPEL functions

This section talks about the important and commonly used functions related to BPEL. These are:

- `bpel:getVariableProperty()`
- `bpel:doXslTransform()`

getVariableProperty()

This function is used to extract global property values from variables.

Syntax

```
bpel:getVariableProperty ('variableName', 'propertyName')
```

Example

```
bpel:getVariableProperty('TicketApproval', 'Class')
```

doXslTransform()

We can invoke an XSLT transformation from an assignment using the `bpel:doXslTransform()` function.

Syntax

```
bpel:doXslTransform('style-sheet-URI', node-set,
                    ('xslt-parameter', value)*)
```

Example

```
<assign>
  <copy>
    <from>
        bpel:doXslTransform("http://packtpub.com/xslt/person.xsl",
                            $PersonData)
    </from>
    <to variable="InsuredPersonRequest" />
  </copy>
</assign>
```

Deadline and duration expressions

To specify deadlines and durations for activities, BPEL uses lexical representations of corresponding XML Schema data types. For setting deadlines, the data types are either `dateTime` or `date`. For setting the duration (a timeout, for instance), you can use the `duration` data type. The lexical representation of expressions should conform to the XPath 1.0 (or the selected query language) specifications. Such expressions should evaluate to the values of corresponding XML Schema types—`dateTime` and `date` for deadline and `duration` for duration expressions.

All three data types use lexical representations that conform to the ISO 8601 standard. For more information on this standard, see the ISO web page at `http://www.iso.ch`. The ISO 8601 lexical format uses characters within date and time information. Characters are appended to the numbers and have the following meaning:

- `C` represents centuries.
- `Y` represents years.
- `M` represents months.
- `D` represents days.
- `h` represents hours.
- `m` represents minutes.
- `s` represents seconds. Seconds can be represented in the format `ss.sss` to increase precision.
- `Z` is used to designate **Coordinated Universal Time** (**UTC**). It should immediately follow the time-of-day element.

For the `dateTime` expressions, there is an additional designator:

- `T` is used as a time designator that indicates the start of the representation of the time.

For `duration` expressions, the following characters can also be used:

- `P` is used as the time duration designator. Duration expressions always start with `P`.
- `Y` represents the number of years.
- `M` represents the number of months or minutes.
- `D` represents the number of days.
- `H` represents the number of hours.
- `S` represents the number of seconds.

Standard elements

Each activity has standard optional nested elements `<sources>` and `<targets>`.

Syntax

```
<targets>?
  <joinCondition expressionLanguage="anyURI"?>?
    bool-expr
  </joinCondition>
  <target linkName="NCName" />+
</targets>

<sources>?
  <source linkName="NCName">+
    <transitionCondition expressionLanguage="anyURI"?>?
        bool-expr
    </transitionCondition>
  </source>
</sources>
```

Standard attributes

Each activity has two standard optional attributes:

- Name
- An indicator to state whether a join fault should be suppressed if it occurs

Syntax

The syntax associated with these attributes within the properties is as follows:

```
name="NCName"?
suppressJoinFailure="yes|no"?
```

Default values of attributes

In the following table, we cover the default values of some BPEL attributes.

Attribute	Default Value
`createInstance`	`no`
`exitOnStandardFault`	`no`
`expressionLanguage`	`urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0`
`initializePartnerRole`	`no`
`initiate`	`no`
`isolated`	`no`
`keepSrcElementName`	`no`
`queryLanguage`	`urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0`
`successfulBranchesOnly`	`no`
`suppressJoinFailure`	`no`
`validate`	`no`

Standard faults

The following table is a list of standard faults that commonly arise in BPEL execution.

Fault name	Fault is thrown:
`ambiguousReceive`	Thrown when a business process instance simultaneously enables two or more IMAs for the same partner link, port type, and operation but different `correlationSets`, and the correlations of multiple of these activities match an incoming request message.
`completionConditionFailure`	Thrown upon completion of a directly enclosed `<scope>` activity within `<forEach>` activity it can be determined that the completion condition can never be true.
`conflictingReceive`	Thrown when more than one inbound message activity is enabled simultaneously for the same partner link, port type, operation, and correlation set(s).
`conflictingRequest`	Thrown when more than one inbound message activity is open for the same partner link, operation, and message exchange.

Fault name	Fault is thrown:
`correlationViolation`	Thrown when the contents of the messages that are processed in an `<invoke>`, `<receive>`, `<reply>`, `<onMessage>`, or `<onEvent>` do not match specified correlation information.
`invalidBranchCondition`	Thrown if the integer value used in the `<branches>` completion condition of `<forEach>` is larger than the number of directly enclosed `<scope>` activities.
`invalidExpressionValue`	Thrown when an expression used within a WS-BPEL construct (except `<assign>`) returns an invalid value with respect to the expected XML Schema type.
`invalidVariables`	Thrown when an XML Schema validation (implicit or explicit) of a variable value fails.
`joinFailure`	Thrown when the join condition of an activity evaluates to `false` and the value of the `suppressJoinFailure` attribute is `yes`.
`mismatchedAssignmentFailure`	Thrown when incompatible types or incompatible XML infoset structure are encountered in an `<assign>` activity.
`missingReply`	Thrown when an inbound message activity has been executed, and the process instance or scope instance reaches the end of its execution without a corresponding `<reply>` activity having been executed.
`missingRequest`	Thrown when a `<reply>` activity cannot be associated with an open inbound message activity by matching the partner link, operation, and message exchange tuple.
`scopeInitializationFailure`	Thrown if there is any problem creating any of the objects defined as part of scope initialization. This fault is always caught by the parent scope of the faulted scope.
`selectionFailure`	Thrown when a selection operation performed either in a function such as `bpel:getVariableProperty`, or in an assignment, encounters an error.
`subLanguageExecutionFault`	Thrown when the execution of an expression results in an unhandled fault in an expression language or query language.

Fault name	Fault is thrown:
uninitializedPartnerRole	Thrown when an <invoke> or <assign> activity references a partner link whose partnerRole endpoint reference is not initialized.
uninitializedVariable	Thrown when there is an attempt to access the value of an uninitialized variable or, in the case of a message type variable, one of its uninitialized parts.
unsupportedReference	Thrown when a WS-BPEL implementation fails to interpret the combination of the reference-scheme attribute and the content element OR just the content element alone.
xsltInvalidSource	Thrown when the transformation source provided in a bpel:doXslTransform function call was not legal (that is, not an EII).
xsltStylesheetNotFound	Thrown when the named stylesheet in a bpel:doXslTransform function call was not found.

Namespaces

In the following table, the BPEL and partner link namespaces are listed.

Prefix	Namespace URL
abstract	http://docs.oasis-open.org/wsbpel/2.0/process/abstract
bpel	http://docs.oasis-open.org/wsbpel/2.0/process/executable
plnk	http://docs.oasis-open.org/wsbpel/2.0/plnktype
sref	http://docs.oasis-open.org/wsbpel/2.0/serviceref
vprop	http://docs.oasis-open.org/wsbpel/2.0/varprop
wsdl	http://schemas.xmlsoap.org/wsdl/
xsd	http://www.w3.org/2001/XMLSchema
xsi	http://www.w3.org/2001/XMLSchema-instance