13 BPMN Execution Semantics

13.1 General

NOTE: The content of this clause is REQUIRED for BPMN Process Execution Conformance or for BPMN Complete Conformance. However, this clause is NOT REQUIRED for BPMN Process Modeling Conformance, BPMN Choreography Conformance, or BPMN BPEL Process Execution Conformance. For more information about BPMN conformance types, see page 1.

This sub clause defines the execution semantics for orchestrations in **BPMN 2.0.2**. The purpose of this execution semantics is to describe a clear and precise understanding of the operation of the elements. However, for some elements only conceptual model is provided which does not specify details needed to execute them on an engine. These elements are called non-operational. Implementations MAY extend the semantics of non-operational elements to make them executable, but this is considered to be an optional extension to **BPMN**. Non-operational elements MAY be ignored by implementations conforming to **BPMN Process Execution Conformance** type. The following elements are non-operational:

- **Manual Task**
- **Abstract Task**
- `DataState`
- `IORules`
- **Ad-Hoc Process**
- `ItemDefinitions` with an `itemKind` of `Physical`
- the `inputSetWithWhileExecuting` attribute of **DataInput**
- the `outputSetWithWhileExecuting` attribute of **DataOutput**
- the `isClosed` attribute of **Process**
- the `isImmediate` attribute of **Sequence Flow**

The execution semantics are described informally (textually), and this is based on prior research involving the formalization of execution semantics using mathematical formalisms.

This sub clause provides the execution semantics of elements through the following structure:

- A description of the operational semantics of the element.
- Exception issues for the element where relevant.
- List of workflow patterns[1] supported by the element where relevant.

1. http://www.workflowpatterns.com/patterns/control/index.php

13.2 Process Instantiation and Termination

A **Process** is instantiated when one of its **Start Events** occurs. Each occurrence of a **Start Event** creates a new **Process** *Instance* unless the **Start Event** participates in a **Conversation** that includes other **Start Events**. In that case, a new **Process** *instance* is only created if none already exists for the specific **Conversation** (identified through its associated correlation information) of the **Event** occurrence. Subsequent **Start Events** that share the same correlation information as a **Start Event** that created a **Process** *instance* are routed to that **Process** *instance*. Note that a *global* **Process** MUST neither have any empty **Start Event** nor any **Gateway** or **Activity** without *incoming* **Sequence Flows**. An exception is the **Event Gateway**.

A **Process** can also be started via an **Event-Based Gateway** or a **Receive Task** that has no *incoming* **Sequence Flows** and its `instantiate` flag set to *true*. If the **Event-Based Gateway** is *exclusive*, the first matching **Event** will create a new *instance* of the **Process**. The **Process** then does not wait for the other **Events** originating from the same **Event-Based Gateway** (see also semantics of the **Event-Based Exclusive Gateway** on page 437). If the **Event-Based Gateway** is *parallel*, also the first matching **Event** creates a new **Process** *instance*. However, the **Process** then waits for the other **Events** to arrive. As stated above, those **Events** MUST have the same correlation information as the **Event** that arrived first. A **Process** *instance* completes only if all **Events** that succeed a **Parallel Event-Based Gateway** have occurred.

To specify that the instantiation of a **Process** waits for multiple **Start Events** to happen, a **Multiple Parallel Start Event** can be used.

Note that two **Start Events** are alternative, A **Process** *instance* triggered by one of the **Start Events** does not wait for an alternative **Start Event** to occur. Note that there MAY be multiple instantiating **Parallel Event-Based Gateways**. This allows the modeler to express that either all the **Events** after the first **Gateway** occur or all the **Events** after the second **Gateway** and so forth.

Each **Start Event** that occurs creates a *token* on its *outgoing* **Sequence Flows**, which is followed as described by the semantics of the other **Process** elements.

- A **Process** *instance* is completed, if and only if the following three conditions hold:
  - If the *instance* was created through an instantiating **Parallel Gateway**, then all subsequent **Events** (of that **Gateway**) MUST have occurred.
  - There is no *token* remaining within the **Process** *instance*.
  - No **Activity** of the **Process** is still active.

For a **Process** *instance* to become completed, all *tokens* in that *instance* MUST reach an end node, i.e., a node without *outgoing* **Sequence Flows**. A *token* reaching an **End Event** triggers the behavior associated with the **Event** type, e.g., the associated **Message** is sent for a **Message End Event**, the associated *Signal* is sent for a **Signal End Event**, and so on. If a *token* reaches a **Terminate End Event**, the entire **Process** is abnormally terminated.

13.3 Activities

This sub clause specifies the semantics of **Activities**. First the semantics that is common to all **Activities** is described. Subsequently the semantics of special types of **Activities** is described.

13.3.1 Sequence Flow Considerations

The nature and behavior of **Sequence Flows** is described in “Sequence Flow” on page 95. But there are special considerations relative to **Sequence Flows** when applied to **Activities**. An **Activity** that is the target of multiple **Sequence Flows** participates in “uncontrolled flow.”

To facilitate the definition of **Sequence Flow** (and other **Process** elements) behavior, we employ the concept of a *token* that will traverse the **Sequence Flows** and pass through the elements in the **Process**. A *token* is a theoretical concept that is used as an aid to define the behavior of a **Process** that is being performed. The behavior of **Process** elements can be defined by describing how they interact with a *token* as it “traverses” the structure of the **Process**. However, modeling and execution tools that implement **BPMN** are NOT REQUIRED to implement any form of *token*.

Uncontrolled flow means that, for each *token* arriving on any *incoming* **Sequence Flows** into the **Activity**, the **Task** will be enabled independently of the arrival of *tokens* on other *incoming* **Sequence Flows**. The presence of multiple *incoming* **Sequence Flows** behaves as an **exclusive gateway**. If the flow of *tokens* into the **Task** needs to be ‘controlled,’ then **Gateways** (other than **Exclusive**) should be explicitly included in the **Process** flow prior to the **Task** to fully eliminate semantic ambiguities.

If an **Activity** has no *incoming* **Sequence Flows**, the **Activity** will be instantiated when the containing **Process** or **Sub-Process** is instantiated. Exceptions to this are **Compensation Activities**, as they have specialized instantiation behavior.

Activities can also be source of **Sequence Flows**. If an **Activity** has multiple *outgoing* **Sequence Flows**, all of them will receive a *token* when the **Activity** transitions to the *Completed* state. Semantics for *token* propagation for other termination states is defined below. Thus, multiple *outgoing* **Sequence Flows** behaves as a parallel split. Multiple *outgoing* **Sequence Flows** with conditions behaves as an inclusive split. A mix of multiple *outgoing* **Sequence Flows** with and without conditions is considered as a combination of a parallel and an inclusive split as shown in the Figure 13.1.

Figure 13.1 – Behavior of multiple outgoing Sequence Flows of an Activity

If the **Activity** has no *outgoing* **Sequence Flows**, the **Activity** will terminate without producing any *tokens* and termination semantics for the container is then applied.

Token movement across a **Sequence Flow** does not have any timing constraints. A *token* might take a long or short time to move across the **Sequence Flow**. If the `isImmediate` attribute of a **Sequence Flow** has a value of *false*, or has no value and is taken to mean *false*, then **Activities** not in the model MAY be executed while the *token* is moving along the **Sequence Flow**. If the `isImmediate` attribute of a **Sequence Flow** has a value of *true*, or has no value and is taken to mean *true*, then **Activities** not in the model MAY NOT be executed while the *token* is moving along the **Sequence Flow**.

13.3.2 Activity

An **Activity** is a **Process** step that can be atomic (**Tasks**) or decomposable (**Sub-Processes**) and is executed by either a system (automated) or humans (manual). All **Activities** share common attributes and behavior such as states and state transitions. An **Activity**, regardless of type, has lifecycle generally characterizing its operational semantics. The lifecycle, described as a UML state diagram in Figure 13.2, entails states and transitions between the states.

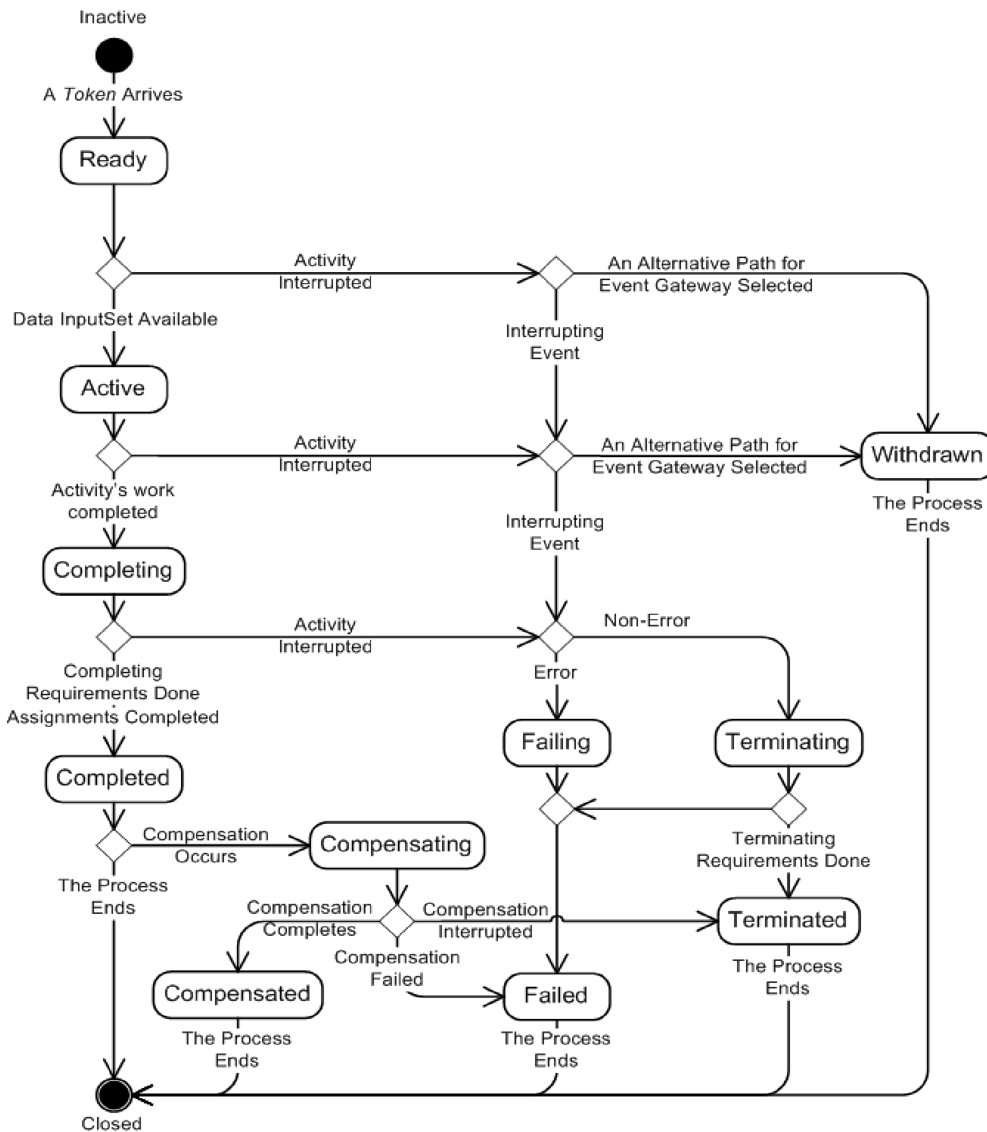


Figure 13.2 – The Lifecycle of a BPMN Activity

The lifecycle of an **Activity** is described as follows:

- An **Activity** is *Ready* for execution if the REQUIRED number of *tokens* is available to activate the **Activity**. The REQUIRED number of *tokens* (one or more) is indicated by the attribute **StartQuantity**. If the **Activity** has more than one *Incoming* **Sequence Flows**, there is an implied **Exclusive Gateway** that defines the behavior.
- When some data `InputSet` becomes available, the **Activity** changes from *Ready* to the *Active* state. The availability of a data `InputSet` is evaluated as follows. The data `InputSets` are evaluated in order. For each `InputSet`, the data inputs are filled with data coming from the elements of the context such as **Data Objects** or `Properties` by triggering the input **Data Associations**. An `InputSet` is *available* if each of its REQUIRED **Data Inputs** is available. A data input is REQUIRED by a data `InputSet` if it is not optional in that `InputSet`. If an `InputSet` is available, it is used to start the **Activity**. Further `InputSets` are not evaluated. If an `InputSet` is not available, the next `InputSet` is evaluated. The **Activity** waits until one `InputSet` becomes available. Please refer to 10.4.2 on page 224 for a description of the execution semantics for **Data Associations**.
- An **Activity**, if *Ready or Active*, can be *Withdrawn* from being able to complete in the context of a race condition. This situation occurs for **Tasks** that are attached after an **Event-Based Exclusive Gateway**. The first element (**Task** or **Event**) that completes causes all other **Tasks** to be withdrawn.
- If an **Activity** fails during execution, it changes from the state *Active* to *Failed*.
    - If a fault happens in the environment of the **Activity**, termination of the **Activity** is triggered, causing the **Activity** to go into the state *Terminated*.
- If an **Activity**'s execution ends without anomalies, the **Activity**'s state changes to *Completing*. This intermediate state caters for processing steps prior to completion of the **Activity**. An example of where this is useful is when non-interrupting *Event Handlers* (proposed for **BPMN 2.0**) are attached to an **Activity**. They need to complete before the **Activity** to which it is attached can complete. The state *Completing* of the main **Activity** indicates that the execution of the main **Activity** has been completed, however, the main **Activity** is not allowed to be in the state *Completed*, as it still has to wait for all non-interrupting *Event Handlers* to complete. The state Completing does not allow further processing steps, otherwise allowed during the execution of the **Activity**. For example, new attached non-interrupting *Event Handlers* MAY be created as long as the main **Activity** is in state *Active*. However, once in the state *Completing*, running handlers should be completed with no possibility to create new ones.
- An **Activity**'s execution is interrupted if an interrupting **Event** is raised (such as an *error*) or if an interrupting **Event Sub-Process** is initiated, In this case, the **Activity**'s state changes to *Failing* (in case of an *error*) or *Terminating* (in case any other interrupting **Event**). All nested **Activities** that are not in *Ready*, *Active* or a final state (*Completed*, *Compensated*, *Failed*, etc.) and non-interrupting **Event Sub-Processes** are terminated. The data context of the **Activity** is preserved in case an interrupting **Event Sub-Process** is invoked. The data context is released after the **Event Sub-Process** reaches a final state.
- After all completion dependencies have been fulfilled, the state of the **Activity** changes to *Completed*. The *outgoing* **Sequence Flows** becomes active and a number of *tokens*, indicated by the attribute **CompletionQuantity**, is placed on it. If there is more than one outbound **Sequence Flows** for an **Activity**, it behaves like an implicit **Parallel Gateway**. Upon completion, also a data `OutputSet` of the **Activity** is selected as follows. All `OutputSets` are checked for availability in order. An `OutputSet` is available if all its REQUIRED **Data Outputs** are available. A data output is REQUIRED by an `OutputSet` if it is not optional in that `OutputSet`. If the data `OutputSet` is available, data is pushed into the context of the **Activity** by triggering the output **Data Associations** of all its data outputs. Further `OutputSets` are not evaluated. If the data `OutputSet` is not available, the next data `OutputSet` is checked. If no `OutputSet` is available, a runtime exception is thrown. If the **Activity** has an associated `IORule`, the chosen `OutputSet` is checked against that `IORule`, i.e., it is checked whether the `InputSet` that was used in starting the **Activity** *instance* is together with the chosen `OutputSet` compliant with the `IORule`. If not, a runtime exception is thrown.

- Only completed **Activities** could, in principle, be compensated, however, the **Activity** can end in state *Completed*, as *compensation* might not be triggered or there might be no *compensation handler* specified. If the *compensation handler* is invoked, the **Activity** changes to state *Compensating* until either *compensation* finishes successfully (state *Compensated*), an exception occurs (state *Failed*), or controlled or uncontrolled termination is triggered (state *Terminated*).

13.3.3 Task

Task execution and completion for the different **Task** types are as follows:

- **Service Task**: Upon activation, the data in the `inMessage` of the `Operation` is assigned from the data in the **Data Input** of the **Service Task** the `Operation` is invoked. On completion of the service, the data in the **Data Output** of the **Service Task** is assigned from the data in the `outMessage` of the `Operation`, and the **Service Task** completes. If the invoked service returns a *fault*, that *fault* is treated as interrupting *error*, and the **Activity** fails.
- **Send Task**: Upon activation, the data in the associated **Message** is assigned from the data in the **Data Input** of the **Send Task**. The **Message** is sent and the **Send Task** completes.
- **Receive Task**: Upon activation, the **Receive Task** begins waiting for the associated **Message**. When the **Message** arrives, the data in the **Data Output** of the **Receive Task** is assigned from the data in the **Message**, and **Receive Task** completes. For key-based *correlation*, only a single receive for a given `CorrelationKey` can be active, and thus the **Message** matches at most one **Process** *instance*. For predicate-based *correlation*, the **Message** can be passed to multiple **Receive Tasks**. If the **Receive Task**'s `instantiate` attribute is set to *true*, the **Receive Task** itself can start a new **Process** *instance*.
- **User Task**: Upon activation, the **User Task** is distributed to the assigned person or group of people. When the work has been done, the **User Task** completes.
- **Manual Task**: Upon activation, the manual task is distributed to the assigned person or group of people. When the work has been done, the **Manual Task** completes. This is a conceptual model only; a **Manual Task** is never actually executed by an IT system.
- **Business Rule Task**: Upon activation, the associated business rule is called. On completion of the business rule, the **Business Rule Task** completes.
- **Script Task**: Upon activation, the associated script is invoked. On completion of the script, the **Script Task** completes.
- **Abstract Task**: Upon activation, the **Abstract Task** completes. This is a conceptual model only; an **Abstract Task** is never actually executed by an IT system.

13.3.4 Sub-Process/Call Activity

A **Sub-Process** is an **Activity** that encapsulates a **Process** that is in turn modeled by **Activities**, **Gateways**, **Events**, and **Sequence Flows**. Once a **Sub-Process** is instantiated, its elements behave as in a normal **Process**. The instantiation and completion of a **Sub-Process** is defined as follows:

- A **Sub-Process** is instantiated when it is reached by a **Sequence Flow** *token*. The **Sub-Process** has either a unique empty **Start Event**, which gets a *token* upon instantiation, or it has no **Start Event** but **Activities** and **Gateways** without *incoming* **Sequence Flows**. In the latter case all such **Activities** and **Gateways** get a *token*. A **Sub-Process** MUST not have any non-empty **Start Events**.

- If the **Sub-Process** does not have incoming **Sequence Flows** but **Start Events** that are target of **Sequence Flows** from outside the **Sub-Process**, the **Sub-Process** is instantiated when one of these **Start Events** is reached by a *token*. Multiple such **Start Events** are alternative, i.e., each such **Start Event** that is reached by a *token* generates a new *instance*.
- A **Sub-Process** *instance* completes when there are no more *tokens* in the **Sub-Process** and none of its **Activities** is still active.
- If a "terminate" **End Event** is reached, the **Sub-Process** is abnormally terminated. For a "cancel" **End Event**, the **Sub-Process** is abnormally terminated and the associated *Transaction* is aborted. Control leaves the **Sub-Process** through a cancel intermediate boundary **Event**. For all other **End Events**, the behavior associated with the **Event** type is performed, e.g., the associated **Message** is sent for a **Message End Event**, the associated signal is sent for a signal **End Event**, and so on.
- If a global **Process** is called through a **Call Activity**, then the **Call Activity** has the same instantiation and termination semantics as a **Sub-Process**. However, in contrast to a **Sub-Process**, the global **Process** that is called MAY also have non-empty **Start Events**. These non-empty **Start Events** are alternative to the empty **Start Event** and hence they are ignored when the **Process** is called from another **Process**.

13.3.5 Ad-Hoc Sub-Process

An **Ad-Hoc Sub-Process** or **Process** contains a number of embedded inner **Activities** and is intended to be executed with a more flexible ordering compared to the typical routing of **Processes**. Unlike regular **Processes**, it does not contain a complete, structured **BPMN** diagram description—i.e., from **Start Event** to **End Event**. Instead the **Ad-Hoc Sub-Process** contains only **Activities**, **Sequence Flows**, **Gateways**, and **Intermediate Events**. An **Ad-Hoc Sub-Process** MAY also contain **Data Objects** and **Data Associations**. The **Activities** within the **Ad-Hoc Sub-Process** are not REQUIRED to have *incoming* and *outgoing* **Sequence Flows**. However, it is possible to specify **Sequence Flows** between some of the contained **Activities**. When used, **Sequence Flows** will provide the same ordering constraints as in a regular **Process**. To have any meaning, **Intermediate Events** will have *outgoing* **Sequence Flows** and they can be triggered multiple times while the **Ad-Hoc Sub-Process** is active.

The contained **Activities** are executed sequentially or in parallel, they can be executed multiple times in an order that is only constrained through the specified **Sequence Flows**, **Gateways**, and data connections.

Operational semantics

- At any point in time, a subset of the embedded **Activities** is *enabled*. Initially, all **Activities** without *incoming* **Sequence Flows** are enabled. One of the enabled **Activities** is selected for execution. This is not done by the implementation but usually by a *Human Performer*. If the `ordering` attribute is set to sequential, another enabled **Activity** can be selected for execution only if the previous one has terminated. If the `ordering` attribute is set to parallel, another enabled **Activity** can be selected for execution at any time. This implies the possibility of the multiple parallel *instances* of the same inner **Activity**.
- After each completion of an inner **Activity**, a condition specified through the `completionCondition` attribute is evaluated:
  - If *false*, the set of enabled inner **Activities** is updated and new **Activities** can be selected for execution.
  - If *true*, the **Ad-Hoc Sub-Process** completes without executing further inner **Activities**. In case the ordering attribute is set to parallel and the attribute `cancelRemainingInstances` is *true*, running *instances* of inner **Activities** are canceled. If `cancelRemainingInstances` is set to *false*, the **Ad-Hoc Sub-Process** completes after all remaining inner *instances* have completed or terminated.

- When an inner **Activity** with *outgoing* **Sequence Flows** completes, a number of *tokens* are produced on its *outgoing* **Sequence Flows**. This number is specified through its attribute `completionQuantity`. The resulting state MAY contain also other *tokens* on *incoming* **Sequence Flows** of either **Activities**, converging **Parallel** or **Complex Gateways**, or an **Intermediate Event**. Then all *tokens* are propagated as far as possible, i.e., all activated **Gateways** are executed until no **Gateway** and **Intermediate Event** is activated anymore. Consequently, a state is obtained where each *token* is on an *incoming* **Sequence Flow** of either an inner **Activity**, a converging **Parallel** or **Complex Gateway** or an **Intermediate Event**. An inner **Activity** is now enabled if it has either no *incoming* **Sequence Flows** or there are sufficiently many *tokens* on its *incoming* **Sequence Flows** (as specified through `startQuantity`).

Workflow patterns: WCP-17 Interleaved parallel routing.

13.3.6 Loop Activity

The **Loop Activity** is a type of **Activity** that acts as a wrapper for an inner **Activity** that can be executed multiple times in sequence.

Operational semantics: Attributes can be set to determine the behavior. The **Loop Activity** executes the inner **Activity** as long as the loopCondition evaluates to *true*. A testBefore attribute is set to decide when the loopCondition should be evaluated: either *before* the **Activity** is executed or *after*, corresponding to a pre- and post-tested *loop* respectively. A loopMaximum attribute can be set to specify a maximal number of iterations. If it is not set, the number is unbounded.

Workflow Patterns Support: WCP-21 Structured Loop.

13.3.7 Multiple Instances Activity

The *multi-instance* (MI) **Activity** is a type of **Activity** that acts as a wrapper for an **Activity** which has multiple *instances* spawned in parallel or sequentially.

Operational semantics: The MI specific attributes are used to configure specific behavior. The attribute isSequential determines whether *instances* are generated sequentially (*true*) or in parallel (*false*). The number of *instances* to be generated is either specified by the integer-valued `Expression loopCardinality` or as the cardinality of a specific collection-valued data item of the data input of the MI **Activity**. The latter is described in detail below.

The number of *instances* to be generated is evaluated once. Subsequently the number of *instances* are generated. If the *instances* are generated sequentially, a new *instance* is generated only after the previous has been completed. Otherwise, multiple *instances* to be executed in parallel are generated.

Attributes are available to support the different possibilities of behavior. The `completionCondition Expression` is a boolean predicate that is evaluated every time an *instance* completes. When evaluated to *true*, the remaining *instances* are canceled, a *token* is produced for the *outgoing* **Sequence Flows**, and the **MI Activity** completes.

The attribute `behavior` defines if and when an **Event** is thrown from an **Activity** *instance* that is about to complete. It has values of `none`, `one`, `all`, and `complex`, assuming the following behavior:

- **none**: an `EventDefinition` is thrown for all *instances* completing.
- **one**: an `EventDefinition` is thrown upon the first *instance* completing.
- **all**: no **Event** is ever thrown.
- **complex**: the `complexBehaviorDefinitions` are consulted to determine if and which **Events** to throw.

For the behaviors of none and one, an EventDefinition (which is referenced from MultipleInstanceLoopCharacteristics through the noneEvent and oneEvent associations, respectively) is thrown which automatically carries the current runtime attributes of the **MI Activity**. That is, the ItemDefinition of these SignalEventDefinitions is implicitly given by the specific runtime attributes of the **MI Activity**.

The complexBehaviorDefinition association references multiple ComplexBehaviorDefinition entities which each point to a boolean condition being a FormalExpression and an Event which is an ImplicitThrowEvent. Whenever an **Activity** *instance* completes, the conditions of all ComplexBehaviorDefinitions are evaluated. For each ComplexBehaviorDefinition whose condition is evaluated to *true*, the associated **Event** is automatically thrown. That is, a single **Activity** completion can lead to multiple different **Events** that are thrown. The **Events** can then be caught on the boundary of the **MI Activity**. Multiple ComplexBehaviorDefinitions offer an easy way of implicitly spawning different flow at the **MI Activity** boundary for different situations indicating different states of progress in the course of executing the **MI Activity**.

The completionCondition, the condition in the ComplexBehaviorDefinition, and the DataInputAssociation of the **Event** in the ComplexBehaviorDefinition can refer to the **MI Activity** *instance* attributes and the loopDataInput, loopDataOutput, inputDataItem, and outputDataItem that are referenced from the MultiInstanceLoopCharacteristics.

In practice, an MI **Activity** is executed over a data *collection*, processing as input the data values in the *collection* and producing as *output* data values in a *collection*. The *input* data collection is passed to the MI outer **Activity**'s loopDataInput from a **Data Object** in the **Process** *scope* of the **MI Activity**. Under **BPMN** data flow constraints, the **Data Object** is linked to **MI activity**'s loopDataInput through a DataInputAssociation. To indicate that the **Data Object** is a *collection*, its respective symbol is marked with the MI indicator (three-bar). The items of the loopDataInput *collection* are used to determine the number of *instances* REQUIRED to be executed (whether sequentially or in parallel). Accordingly, the inner *instances* are created and data values from the loopDataInput are extracted and assigned to the respective *instances*. Specifically, the values from the loopDataInput items are passed to an inputDataItem, created in the scope of the outer **Activity**. The value in the inputDataItem can be passed to the loopDataInput of each inner *instance*, where a DataInputAssociation links both. The process of extraction is left under-specified. In practice, it would entail a special-purpose mediator that not only provides the extraction and data assignment, but also any necessary data transformation.

Each *instance* processes the data value of its DataInput. It produces a value in its DataOutput if it completes successfully. The DataOutPut value of the *instance* is passed to a corresponding outputDataItem in the outer **Activity**, where a DataOutputAssociation links both. Each outputDataItem value is updated in the loopDataOutput *collection*, in the corresponding item. The mechanism of this update is left underspecified, and again would be implemented through a special purpose mediator. The loopDataOutput is passed to the **MI Activity**'s **Process** *scope* through a **Data Object** that has a DataOutputAssociation linking both.

It should be noted that the *collection* in the **Process** *scope* should not be accessible until all its items have been written to. This is because, it could be accessed by an **Activity** running concurrently, and therefore control flow through *token* passing cannot guarantee that the *collection* is fully written before it is accessed.

The **MI Activity** is *compensated* only if all its *instances* have completed successfully.

Workflow Patterns Support: WCP-21 Structured Loop, Multiple Instance Patterns WCP 13, 14, 34, 36

13.4 Gateways

This sub clause describes the behavior of **Gateways**.

13.4.1 Parallel Gateway (Fork and Join)

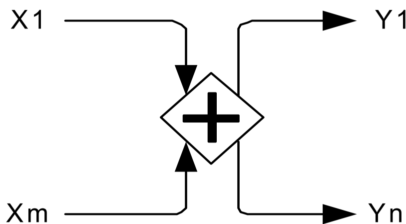


Figure 13.3 – Merging and Branching Sequence Flows for a Parallel Gateway

On the one hand, the **Parallel Gateway** is used to synchronize multiple concurrent branches (merging behavior). On the other hand, it is used to spawn new concurrent threads on parallel branches (branching behavior).

Table 13.1 – Parallel Gateway Execution Semantic

Operational Semantics	The **Parallel Gateway** is activated if there is at least one *token* on each incoming **Sequence Flow**. The **Parallel Gateway** consumes exactly one *token* from each incoming **Sequence Flow** and produces exactly one *token* at each outgoing **Sequence Flow**. If there are excess *tokens* at an incoming **Sequence Flow**, these *tokens* remain at this **Sequence Flow** after execution of the **Gateway**.
Exception Issues	The **Parallel Gateway** cannot throw any exception.
Workflow Patterns Support	Parallel Split (WCP-2) Synchronization (WCP-3)

13.4.2 Exclusive Gateway (Exclusive Decision (data-based) and Exclusive Merge)

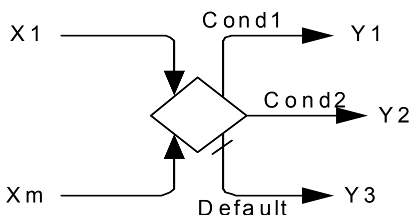


Figure 13.4 – Merging and Branching Sequence Flows for an Exclusive Gateway

The **Exclusive Gateway** has pass-through semantics for a set of incoming branches (merging behavior). Further on, each activation leads to the activation of exactly one out of the set of outgoing branches (branching behavior).

Table 13.2 – Exclusive Gateway Execution Semantics

Operational Semantics	Each *token* arriving at any incoming **Sequence Flows** activates the gateway and is routed to exactly one of the outgoing **Sequence Flows**. In order to determine the outgoing **Sequence Flows** that receives the *token*, the conditions are evaluated in order. The first condition that evaluates to true determines the **Sequence Flow** the *token* is sent to. No more conditions are henceforth evaluated. If and only if none of the conditions evaluates to true, the *token* is passed on the default **Sequence Flow**. In case all conditions evaluate to false and a default flow has not been specified, an exception is thrown.
Exception Issues	The exclusive gateway throws an exception in case all conditions evaluate to false and a default flow has not been specified.
Workflow Patterns Support	Exclusive Choice (WCP-4) Simple Merge (WCP-5) Multi-Merge (WCP-8)

13.4.3 Inclusive Gateway (Inclusive Decision and Inclusive Merge)

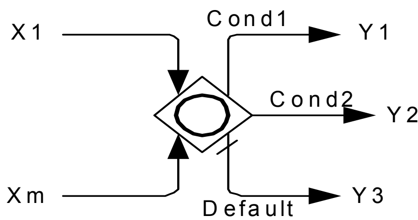


Figure 13.5 – Merging and Branching Sequence Flows for an Inclusive Gateway

The **Inclusive Gateway** synchronizes a certain subset of branches out of the set of concurrent incoming branches (merging behavior). Further on, each firing leads to the creation of threads on a certain subset out of the set of outgoing branches (branching behavior).

Table 13.3 – Inclusive Gateway Execution Semantics

<table>
<tr><td>Operational Semantics</td><td>The Inclusive Gateway is activated if
• At least one incoming Sequence Flow has at least one token and
• For every directed path formed by sequence flow that
- starts with a Sequence Flow f of the diagram that has a token,
- ends with an incoming Sequence Flow of the inclusive gateway that has no token, and
- does not visit the Inclusive Gateway.
• There is also a directed path formed by Sequence Flow that
- starts with f,
- ends with an incoming Sequence Flow of the inclusive gateway that has a token, and
- does not visit the Inclusive Gateway.
Upon execution, a token is consumed from each incoming Sequence Flow that has a token. A token will be produced on some of the outgoing Sequence Flows.
In order to determine the outgoing Sequence Flows that receive a token, all conditions on the outgoing Sequence Flows are evaluated. The evaluation does not have to respect a certain order.
For every condition which evaluates to true, a token MUST be passed on the respective Sequence Flow.
If and only if none of the conditions evaluates to true, the token is passed on the default Sequence Flow.
In case all conditions evaluate to false and a default flow has not been specified, the Inclusive Gateway throws an exception.</td></tr>
<tr><td>Exception Issues</td><td>The inclusive gateway throws an exception in case all conditions evaluate to false and a default flow has not been specified.</td></tr>
<tr><td>Workflow Patterns Support</td><td>Multi-Choice (WCP-6)
Structured Synchronizing Merge (WCP-7)
Acyclic Synchronizing Merge (WCP-37)
General Synchronizing Merge (WCP-38)</td></tr>
</table>

13.4.4 Event-based Gateway (Exclusive Decision (event-based))

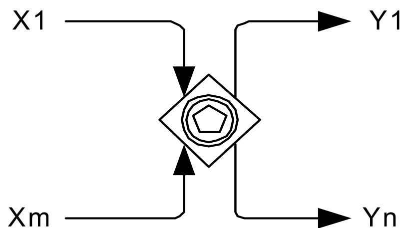


Figure 13.6 – Merging and branching Sequence Flows for an Event-Based Gateway

The **Event-Based Gateway** has pass-through semantics for a set of incoming branches (merging behavior). Exactly one of the outgoing branches is activated afterwards (branching behavior), depending on which of **Events** of the **Gateway** configuration is first triggered. The choice of the branch to be taken is deferred until one of the subsequent **Tasks** or **Events** completes. The first to complete causes all other branches to be withdrawn.

When used at the **Process** start as a **Parallel Event Gateway**, only message-based triggers are allowed. The `Message` *triggers* that are part of the **Gateway** configuration MUST be part of a **Conversation** with the same correlation information. After the first *trigger* instantiates the **Process**, the remaining `Message` *triggers* will be a part of the **Process** *instance* that is already active (rather than creating new **Process** *instances*).

Table 13.4 – Event-Based Gateway Execution Semantics

Exception Issues	The event-based gateway cannot throw any exception.
Workflow Patterns Support	Deferred Choice (WCP-16)

13.4.5 Complex Gateway (related to Complex Condition and Complex Merge)

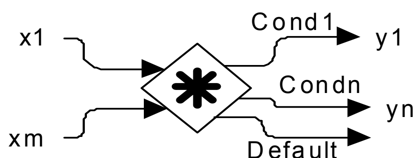


Figure 13.7 – Merging and branching Sequence Flows for a Complex Gateway

The **Complex Gateway** facilitates the specification of complex synchronization behavior, in particular race situations. The diverging behavior is similar to the **Inclusive Gateway**. Each incoming gate of the **Complex Gateway** has an attribute `activationCount`, which can be used in an `Expression` as an integer-valued variable. This variable represents the number of *tokens* that are currently on the respective *incoming* **Sequence Flows**. The **Complex Gateway** has an attribute `activationExpression`. An `activationExpression` is a boolean `Expression` that refers to data and to the `activationCount` of incoming gates. For example, an `activationExpression` could be x1+x2+…+xm >= 3 stating that it needs 3 out of the m incoming gates to have a *token* in order to proceed. To

prevent undesirable oscillation of activation of the **Complex Gateway**, `ActivationCount` variables should only be used in subexpressions of the form *expr* >= *const* where *expr* is an arithmetic `Expression` that uses only addition and *const* is an `Expression` whose evaluation remains constant during execution of the **Process**.

Each *outgoing* **Sequence Flow** of the **Complex Gateway** has a boolean condition that is evaluated to determine whether that **Sequence Flow** receives a *token* during the execution of the **Gateway**. Such a condition MAY refer to internal state of the **Complex Gateway**. There are two states: waiting for start (represented by the runtime attribute `waitingForStart` = *true*) and waiting for reset (`waitingForStart`=*false*).

Table 13.5 – Semantics of the Complex Gateway

Operational Semantics	The **Complex Gateway** is in one of the two states: *waiting for start* or *waiting for reset*, initially it is in *waiting for start*. If it is *waiting for start*, then it waits for the `activationExpression` to become *true*. The `activationExpression` is not evaluated before there is at least one *token* on some *incoming* **Sequence Flow**. When it becomes *true*, a *token* is consumed from each *incoming* **Sequence Flow** that has a *token*. To determine which *outgoing* **Sequence Flow** receive a *token*, all conditions on the *outgoing* **Sequence Flows** are evaluated (in any order). Those and only those that evaluate to *true* receive a *token*. If no condition evaluates to *true*, and only then, the *default* **Sequence Flow** receives a *token*. If no default flow is specified an exception is thrown. The **Gateway** changes its state to *waiting for reset*.The **Gateway** remembers from which of the *incoming* **Sequence Flows** it consumed *tokens* in the first phase. When *waiting for reset*, the **Gateway** waits for a *token* on each of those *incoming* **Sequence Flows** from which it has not yet received a *token* in the first phase unless such a *token* is not expected according to the join behavior of an **inclusive Gateway**. More precisely, the **Gateway** being *waiting for reset*, *resets* when for every directed path formed by sequence flow that - starts with a **Sequence Flow *f*** of the diagram that has a *token*, - ends with an *incoming* **Sequence Flow** of the **Complex Gateway** that has no *token* and has not consumed a *token* in the first phase, and that - does not visit the **Complex Gateway**.

Table13.5 – Semantics of the Complex Gateway

Operational Semantics	• There is also a directed path formed by **Sequence Flow** that - starts with ***f***, - ends with an *incoming* **Sequence Flow** of the **Complex Gateway** that has a *token* or from which a *token* was consumed in the first phase, and that, - does not visit the **Complex Gateway**. If the **Complex Gateway** is contained in a **Sub-Process**, then no paths are considered that cross the boundary of that **Sub-Process**. When the **Gateway** resets, it consumes a *token* from each *incoming* **Sequence Flow** that has a *token* and from which it had not yet consumed a *token* in the first phase. It then evaluates all conditions on the *outgoing* **Sequence Flows** (in any order) to determine which **Sequence Flows** receives a *token*. Those and only those that evaluate to *true* receive a *token*. If no condition evaluates to *true*, and only then, the *default* **Sequence Flow** receives a *token*. The **Gateway** changes its state back to the state *waiting for start*. Note that the **Gateway** might not produce any *tokens* in this phase and no exception is thrown. Note that the conditions on the *outgoing* **Sequence Flows** MAY evaluate differently in the two phases, e.g., by referring to the state of the **Gateway** (runtime attribute `waitingForStart`). Note that if the `activationCondition` never becomes *true* in the first phase, *tokens* are blocked indefinitely at the **Complex Gateway**, which MAY cause a deadlock of the entire **Process**.
Exception issues	The **Complex Gateway** throws an exception when it is activated in the state *waiting for start*, no condition on any *outgoing* **Sequence Flow** evaluates to *true* and no *default* **Sequence Flow** is specified.
Workflow Patterns Support	Structured Discriminator (WCP-9) Blocking Discriminator (WCP-28) Structured Partial Join (WCP-30) Blocking Partial Join (WCP-31)

13.5 Events

This sub clause describes the handling of **Events**.

13.5.1 Start Events

For single **Start Events**, handling consists of starting a new **Process** *instance* each time the **Event** occurs. **Sequence Flows** leaving the **Event** are then followed as usual.

If the **Start Event** participates in a **Conversation** that includes other **Start Events**, a new **Process** *instance* is only created if none already exists for the specific **Conversation** (identified through its associated correlation information) of the **Event** occurrence.

A **Process** can also be started via an **Event-Based Gateway**. In that case, the first matching **Event** will create a new *instance* of the **Process**, and waiting for the other **Events** originating from the same decision stops, following the usual semantics of the **Event-Based Exclusive Gateway**. Note that this is the only scenario where a **Gateway** can exist without *incoming* **Sequence Flows**.

It is possible to have multiple groups of **Event-Based Gateways** starting a **Process**, provided they participate in the same **Conversation** and hence share the same correlation information. In that case, one **Event** out of each group needs to arrive; the first one creates a new **Process** *instance*, while the subsequent ones are routed to the existing *instance*, which is identified through its correlation information.

13.5.2 Intermediate Events

For **Intermediate Events**, the handling consists of waiting for the **Event** to occur. Waiting starts when the **Intermediate Event** is reached. Once the **Event** occurs, it is consumed. **Sequence Flows** leaving the **Event** are followed as usual. For *catch* **Message Intermediate Events**, the **Message** *correlation* behavior is the same as for **Receive Tasks** (see sub clause 13.3.3).

13.5.3 Intermediate Boundary Events

For boundary **Events**, handling first consists of consuming the **Event** occurrence. If the `cancelActivity` attribute is set, the **Activity** the **Event** is attached to is then cancelled (in case of a multi-instance, all its *instances* are cancelled); if the attribute is not set, the **Activity** continues execution (only possible for **Message**, **Signal**, **Timer**, and **Conditional Events**, not for **Error Events**). Execution then follows the **Sequence Flow** connected to the boundary **Event**. For *boundary* **Message Intermediate Events**, the **Message** *correlation* behavior is the same as for **Receive Tasks** (see sub clause 13.3.3).

13.5.4 Event Sub-Processes

Event Sub-Processes allow to handle an **Event** within the context of a given **Sub-Processes** or **Process**. An **Event Sub-Process** always begins with a **Start Event**, followed by **Sequence Flows**. **Event Sub-Processes** are a special kind of **Sub-Process**: they create a scope and are instantiated like a **Sub-Process**, but they are not instantiated by normal control flow but only when the associated **Start Event** is triggered. **Event Sub-Processes** are self-contained and MUST not be connected to the rest of the **Sequence Flows** in the **Sub-Processes**; also they cannot have attached boundary **Events**. They run in the context of the **Sub-Process**, and thus have access to its context.

An **Event Sub-Process** cancels execution of the enclosing **Sub-Process**, if the `isInterrupting` attribute of its **Start Event** is set; for a multi-instance **Activity** this cancels only the affected *instance*. If the `isInterrupting` attribute is not set (not possible for **Error Event Sub-Processes**), execution of the enclosing **Sub-Process** continues in parallel to the **Event Sub-Process**.

An **Event Sub-Process** can optionally re-trigger the **Event** through which it was triggered, to cause its continuation outside the boundary of the associated **Sub-Process**. In that case the **Event Sub-Process** is performed when the **Event** occurs; then control passes to the boundary **Event**, possibly canceling the **Sub-Process** (including running handlers).

Operational semantics

- An **Event Sub-Process** becomes initiated, and thus *Enabled* and *Running*, through the **Activity** to which it is attached. The *Event Handler* MAY only be initiated after the parent **Activity** is Running.

- More than one non-interrupting *Event Handler* MAY be initiated and they MAY be initiated at different times. There might be multiple instances of the non-interrupting *Event Handler* at a time. For **Event Sub-Processes** triggered by a **Message**, the **Message** *correlation* behavior is the same as for **Receive Tasks** -- see sub clause 13.3.3.
- Only one interrupting Event Handler MAY be initiated for a given `EventDefinition` within the context of the parent **Activity**. Once the interrupting *Event Handler* is started, the parent **Activity** is interrupted and no new *Event Handlers* can be initiated or started.An **Event Sub-Process** completes when all *tokens* have reached an **End Event**, like any other **Sub-Process**. If the parent **Activity** enters the state *Completing*, it remains in that state until all contained active **Event Sub-Processes** have completed. While the parent **Activity** is in the *Completing* state, no new **Event Sub-Processes** can be initiated.
- If an interrupting **Event Sub-Process** is started by an *error*, then the parent **Activity** enters the state *Failing* and remains in this state until the interrupting *Event Handler* reaches a final state. During this time, the running *Event Handler* can access to the context of the parent **Activity**. However, new *Event Handlers* MUST NOT be started.
- Similarly, if an interrupting **Event Sub-Process** is started by a non *error* (e.g., Escalation), then the parent **Activity** enters the state *Terminating* and remains in this state until the interrupting *Event Handler* reaches a final state. During this time, the running *Event Handler* can access to the context of the parent **Activity**. However, new *Event Handlers* MUST NOT be started.

13.5.5 Compensation

Compensation is concerned with undoing steps that were already successfully completed, because their results and possibly side effects are no longer desired and need to be reversed. If an **Activity** is still active, it cannot be compensated, but rather needs to be canceled. Cancellation in turn can result in *compensation* of already successfully completed portions of an active **Activity**, in case of a **Sub-Process**.

Compensation is performed by a *compensation handler*. A *compensation handler* can either be a **Compensation Event Sub-Process** (for a **Sub-Process** or **Process**), or an associated **Compensation Activity** (for any **Activity**). A *compensation handler* performs the steps necessary to reverse the effects of an **Activity**. In case of a **Sub-Process**, its **Compensation Event Sub-Process** has access to **Sub-Process** data at the time of its completion (“snapshot data”).

Compensation is triggered by a *throw* **Compensation Event**, which typically will be raised by an *error handler*, as part of cancellation, or recursively by another *compensation handler*. That **Event** specifies the **Activity** for which *compensation* is to be performed, either explicitly or implicitly.

Compensation Handler

A *compensation handler* is a set of **Activities** that are not connected to other portions of the **BPMN** model. The *compensation handler* starts with a *catch* **Compensation Event**. That *catch* **Compensation Event** either is a boundary **Event**, or, in case of a **Compensation Event Sub-Process**, the handler’s **Start Event**.

A *compensation handler* connected via a boundary **Event** can only perform “black-box” *compensation* of the original **Activity**. This *compensation* is modeled with a specialized **Compensation Activity**.

A **Compensation Event Sub-Process** is contained within a **Process** or **Sub-Processes**. It can access data that is part of its parent, snapshot at the point in time when its parent has completed. A *compensation* **Event Sub-Process** can in particular recursively trigger *compensation* for **Activities** contained in that its parent.

It is possible to specify that a **Sub-Process** can be compensated without having to define the *compensation handler*. The **Sub-Process** attribute `compensable`, when set, specifies that default *compensation* is implicitly defined, which recursively compensates all successfully completed **Activities** within that **Sub-Process**, invoking them in reverse order of their forward execution.

Compensation Triggering

Compensation is triggered using a *throw* **Compensation Event**, which can either be an **Intermediate** or an **End Event**. The **Activity** that needs to be compensated is referenced. If the **Activity** is clear from the context, it doesn't have to be specified and defaults to the current **Activity**. A typical scenario for that is an inline *error handler* of a **Sub-Process** that cannot recover the *error*, and as a result would trigger *compensation* for that **Sub-Process**. If no **Activity** is specified in a "global" context, all completed **Activities** in the **Process** are compensated.

By default, *compensation* is triggered synchronously, that is, the *throw* **Compensation Event** waits for the completion of the triggered *compensation handler*. Alternatively, *compensation* can just be triggered without waiting for its completion, by setting the *throw* **Compensation Event**'s `waitForCompletion` attribute to *false*.

Multiple *instances* typically exist for **Loop** or **Multi-Instance Sub-Processes**. Each of these has its own *instance* of its **Compensation Event Sub-Process**, which has access to the specific snapshot data that was current at the time of completion of that particular *instance*. Triggering *compensation* for the **Multi-Instance Sub-Process** individually triggers *compensation* for all *instances* within the current *scope*. If *compensation* is specified via a boundary *compensation handler*, this boundary *compensation handler* also is invoked once for each *instance* of the **Multi-Instance Sub-Process** in the current *scope*.

Relationship between Error Handling and Compensation

Compensation employs a "presumed abort principle," which has a number of consequences. First, only completed **Activities** are compensated; *compensation* of a failed **Activity** results in an empty operation. Thus, when an **Activity** fails, i.e., is left because an *error* has been thrown, it's the *error handler's* responsibility to ensure that no further *compensation* will be necessary once the *error handler* has completed. Second, if no *error* **Event Sub-Process** is specified for a particular **Sub-Process** and a particular *error*, the default behavior is to automatically call *compensation* for all contained **Activities** of that **Sub-Process** if that *error* occurs, thus ensuring the "presumed abort" invariant.

Operational Semantics

- A **Compensation Event Sub-Process** becomes enabled when its *parent* **Activity** transitions into state *Completed*. At that time, a snapshot of the data associated with the parent **Activity** is taken and kept for later usage by the **Compensation Event Sub-Process**. In case the *parent* **Activity** is a *multi-instance* or *loop*, for each *instance* a separate data snapshot is taken, which is used when its associated **Compensation Event Sub-Process** is triggered.
- When *compensation* is triggered for the *parent* **Activity**, its **Compensation Event Sub-Process** is activated and runs. The original context data of the *parent* **Activity** is restored from the data snapshot. In case the *parent* **Activity** is a multi-instance or loop, for each *instance* the dedicated snapshot is restored and a dedicated **Compensation Event Sub-Process** is activated.
- An associated **Compensation Activity** becomes enabled when the **Activity** it is associated with transitions into state *Completed*. When *compensation* is triggered for that **Activity**, the associated **Compensation Activity** is activated. In case the **Activity** is a multi-instance or loop, the **Compensation Activity** is triggered only once, too, and thus has to compensate the effects of all *instances*.
  - Default compensation ensures that **Compensation Activities** are performed in reverse order of the execution of the original **Activities**, allowing for concurrency when there was no dependency between the original **Activities**. Dependencies between original **Activities** that default compensation MUST consider are the following:
    - A **Sequence Flow** between **Activities** A and B results in compensation of B to be performed before compensation of A.

- A data dependency between **Activities** A and B, e.g., through an `IORules` specification in B referring to data produced by A, results in compensation of B to be performed before compensation of A.
- If A and B are two **Activities** that were active as part of an **Ad-Hoc Sub-Process**, then compensation of B MUST be performed before compensation of A if A completed before B started.
- *Instances* of a loop or sequential multi-instance are compensated in reverse order of their forward completion. *Instances* of a parallel multi-instance can be compensated in parallel.
- If a **Sub-Process** A has a *boundary* **Event** connected to **Activity** B, then compensation of B MUST be performed before compensation of A if that particular **Event** occurred. This also applies to multi-instances and loops.

13.5.6 End Events

Process level end events

For a “terminate” **End Event**, the **Process** is abnormally terminated—no other ongoing **Process** *instances* are affected.

For all other **End Events**, the behavior associated with the **Event** type is performed, e.g., the associated **Message** is sent for a **Message End Event**, the associated signal is sent for a **Signal End Event**, and so on. The **Process** *instance* is then completed, if and only if the following two conditions hold:

- All start nodes of the **Process** have been visited. More precisely, all **Start Events** have been triggered, and for all starting **Event-Based Gateways**, one of the associated **Events** has been triggered.
- There is no *token* remaining within the **Process** *instance*.

Sub-process level end events

For a “terminate” **End Event**, the **Sub-Process** is abnormally terminated. In case of a multi-instance **Sub-Process**, only the affected *instance* is terminated—no other ongoing **Sub-Process** *instances* or higher-level **Sub-Process** or **Process** *instances* are affected.

For a “cancel” **End Event**, the **Sub-Process** is abnormally terminated and the associated transaction is aborted. Control leaves the **Sub-Process** through a cancel intermediate boundary **Event**.

For all other **End Events**, the behavior associated with the **Event** type is performed, e.g., the associated **Message** is sent for a **Message End Event**, the associated signal is sent for a signal **End Event**, and so on. The **Sub-Process** *instance* is then completed, if and only if the following two conditions hold:

- All start nodes of the **Sub-Process** have been visited. More precisely, all **Start Events** have been triggered, and for all starting **Event-Based Gateways**, one of the associated **Events** has been triggered.
- There is no *token* remaining within the **Sub-Process** *instance*.

14 Mapping BPMN Models to WS-BPEL

14.1 General

NOTE: The contents of this clause is REQUIRED for BPMN BPEL Process Execution Conformance or for BPMN Complete Conformance. However, this clause is NOT REQUIRED for BPMN Process Modeling Conformance, BPMN Process Choreography Conformance, or BPMN Process Execution Conformance. For more information about BPMN conformance types, see page 1.

This clause covers a mapping of a **BPMN** model to WS-BPEL that is derived by analyzing the **BPMN** objects and the relationships between these objects.

A **Business Process Diagram** can be made up of a set of (semi-) independent components, which are shown as separate **Pools**, each of which represents an orchestration **Process**. There is not a specific mapping of the diagram itself, but rather, each of these *orchestration* **Processes** maps to an individual WS-BPEL *process*.

Not all **BPMN** *orchestration* **Processes** can be mapped to WS-BPEL in a straight-forward way. That is because **BPMN** allows the modeler to draw almost arbitrary graphs to model control flow, whereas in WS-BPEL, there are certain restrictions such as control-flow being either block-structured or not containing cycles. For example, an unstructured *loop* cannot directly be represented in WS-BPEL.

To map a **BPMN** orchestration **Process** to WS-BPEL it MUST be *sound*, that is it MUST contain neither a *deadlock* nor a *lack of synchronization*. A deadlock is a reachable state of the **Process** that contains a *token* on some **Sequence Flow** that cannot be removed in any possible future. A lack of synchronization is a reachable state of the **Process** where there is more than one *token* on some **Sequence Flow**. For further explanation of these terms, we refer to the literature. To define the structure of **BPMN Processes**, we introduce the following concepts and terminology. The **Gateways** and the **Sequence Flows** of the **BPMN** orchestration **Process** form a directed graph. A *block* of the diagram is a connected sub-graph that is connected to the rest of the graph only through exactly two **Sequence Flows**: exactly one **Sequence Flow** entering the block and exactly one **Sequence Flow** leaving the block. A *block hierarchy* for a **Process** model is a set of blocks of the **Process** model in which each pair of blocks is either nested or disjoint and which contains the maximal block (i.e., the whole **Process** model) A block that is nested in another block B is also called a *subblock* of *B* (cf. Figure 14.1). Each block of the block hierarchy of a given **BPMN** orchestration **Process** has a certain structure (or pattern) that provides the basis for defining the BPEL mapping.

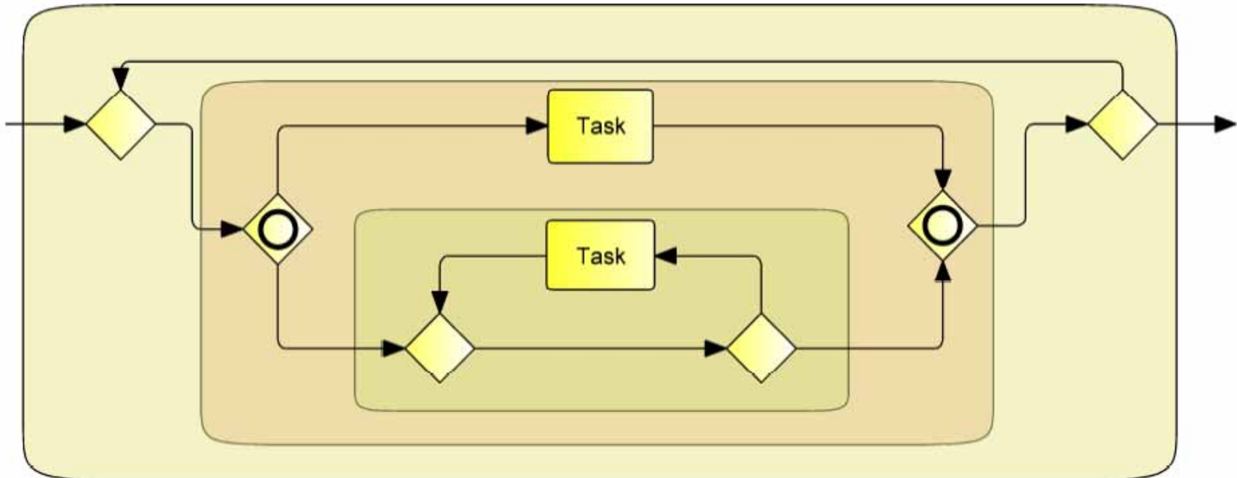


Figure 14.1 – A BPMN orchestration process and its block hierarchy

The following sub clauses define a syntactical BPEL mapping prescribing the resulting BPEL model at the syntactical level, and a semantic BPEL mapping prescribing the resulting BPEL model in terms of its observable behavior. The syntactical BPEL mapping is defined for a subset of **BPMN** models based on certain patterns of **BPMN** blocks, whereas the semantical BPEL mapping (which extends the syntactical mapping) does not enforce block patterns, allowing for the mapping of a larger class of **BPMN** models without prescribing the exact syntactical representation in BPEL.

14.2 Basic BPMN-BPEL Mapping

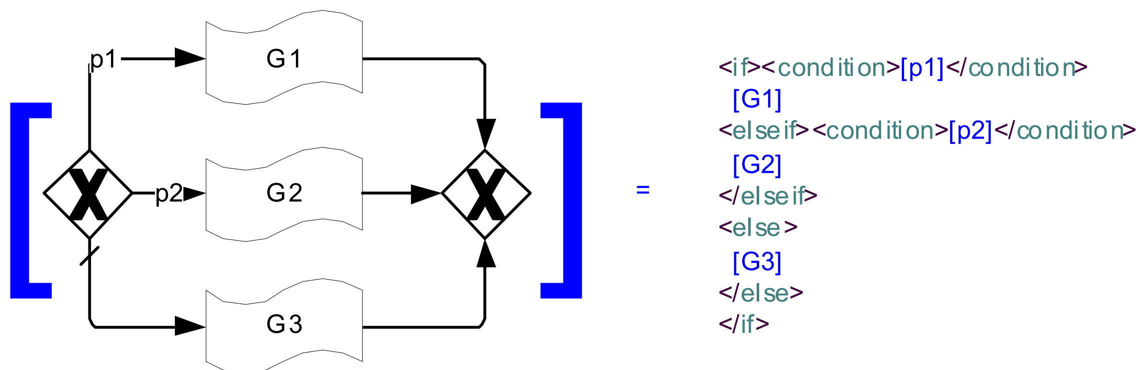
This sub clause introduces a partial mapping function from **BPMN** orchestration **Process** models to WS-BPEL executable **Process** models by recursively defining the mapping for elementary **BPMN** constructs such as **Tasks** and **Events**, and for blocks following the patterns described here. Mapping a **BPMN** block to WS-BPEL includes mapping all of its associated attributes. The observable behavior of a WS-BPEL process resulting from a BPEL mapping is the same as that of the original **BPMN** orchestration **Process**.

We use the notation [**BPMN** construct] to denote the WS-BPEL construct resulting from mapping the **BPMN** construct.

Examples are

[ServiceTask] = Invoke Activity

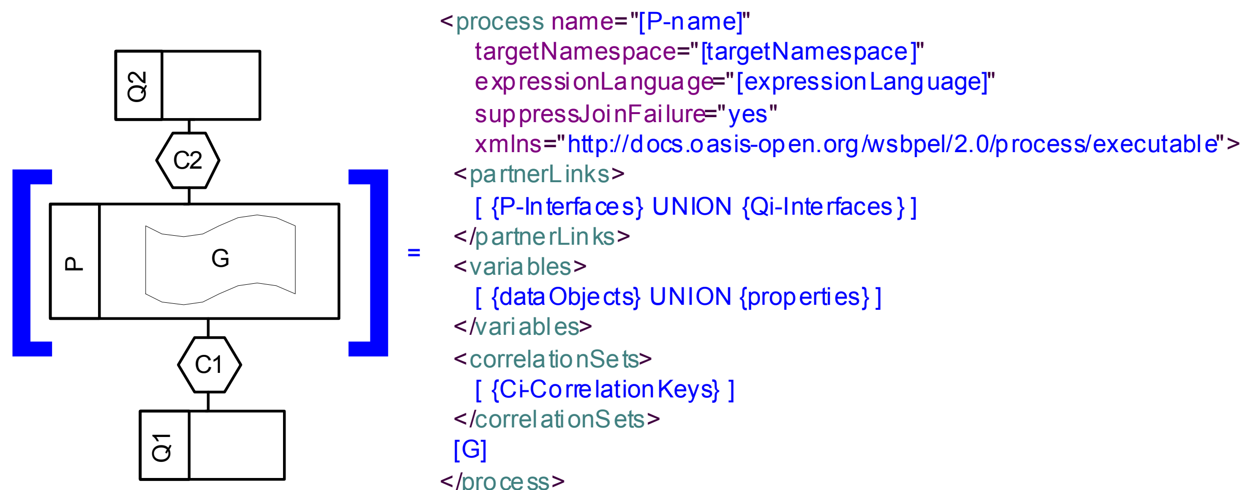
which says that a **BPMN Service Task** is mapped to a WS-BPEL Invoke Activity, or



which says that the data-based exclusive choice controlled by the two predicates p1 and p2, containing the three **BPMN** blocks G1, G2, and G3 is mapped to the WS-BPEL on the right hand side, which recursively uses the mappings of those predicates and those sub-graphs. Note that we use the "waved rectangle" symbol throughout this sub clause to denote **BPMN** blocks.

14.2.1 Process

The following figure describes the mapping of a **Process**, represented by its defining **Collaboration**, to WS-BPEL. The process itself is described by a contained graph G of flow elements to WS-BPEL. The **Process** interacts with *Participants* Q1…Qn via **Conversations** C1…Cm:



The partner links of the corresponding WS-BPEL process are derived from the set of interfaces associated with each participant. Each `interface` of the *Participant* containing the **Process** P itself is mapped to a WS-BPEL partner link with a "myRole" specification, each interface of each other *Participant* Qi is mapped to a WS-BPEL partner link with a "partnerRole" specification.

The variables of the corresponding WS-BPEL process are derived from the set "{dataObjects}" of all **Data Objects** occurring within G, united with the set "{properties}" of all properties occurring within G, without **Data Objects** or properties contained in nested **Sub-Processes**. See "Handling Data" on page 465 for more details of this mapping.

The `correlation sets` of the corresponding WS-BPEL process are derived from the `CorrelationKeys` of the set of **Conversations** C1…Cn (see page 452 for more details of this mapping).

14.2.2 Activities

Common Activity Mappings

The following table displays a set of mappings of general **BPMN Activity** attributes to WS-BPEL activity attributes.

Table 14.1 – Common Activity Mappings to WS-BPEL

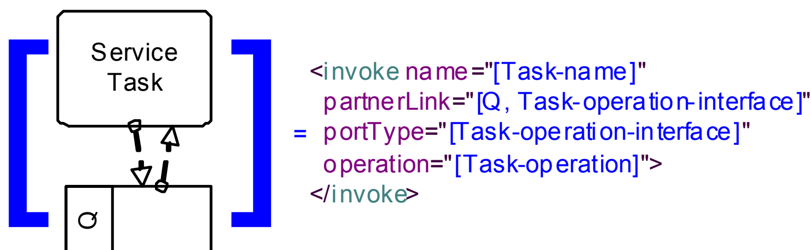
Activity	Mapping to WS-BPEL
name	The name attribute of a BPMN activity is mapped to the name attribute of a WS-BPEL activity by removing all characters not allowed in an XML NCName, and ensuring uniqueness by adding an appropriate suffix. In the subsequent diagrams, this mapping is represented as [name].

Task Mappings

The following sub clauses contain the mappings of the variations of a **Task** to WS-BPEL.

Service Task

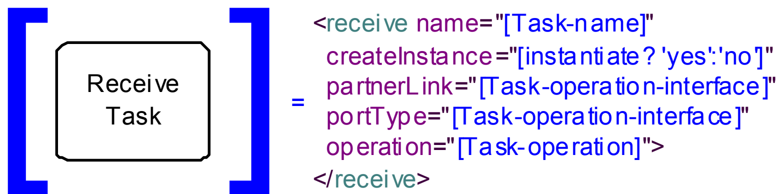
The following figure shows the mapping of a **Service Task** to WS-BPEL.



The partner link associated with the WS-BPEL invoke is derived from both the participant Q that the **Service Task** is connected to by **Message Flows**, and from the `interface` referenced by the `operation` of the **Service Task**.

Receive Task

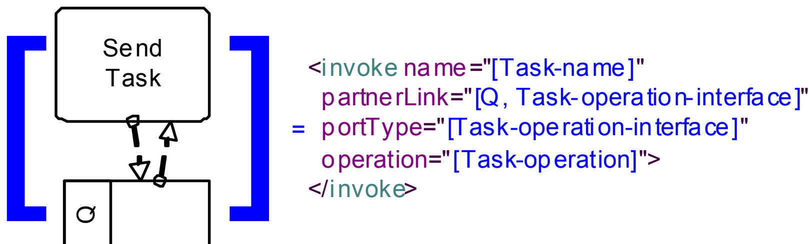
The following figure shows the mapping of a **Receive Task** to WS-BPEL.



The partner link associated with the WS-BPEL receive is derived from the interface referenced by the operation of the **Receive Task**.

Send Task

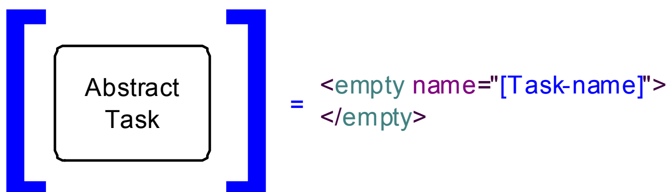
The following figure shows the mapping of a **Send Task** to WS-BPEL.



The partner link associated with the WS-BPEL invoke is derived from both the participant Q that the **Send Task** is connected to by a **Message Flow**, and from the `interface` referenced by the `operation` of the **Send Task**.

Abstract Task

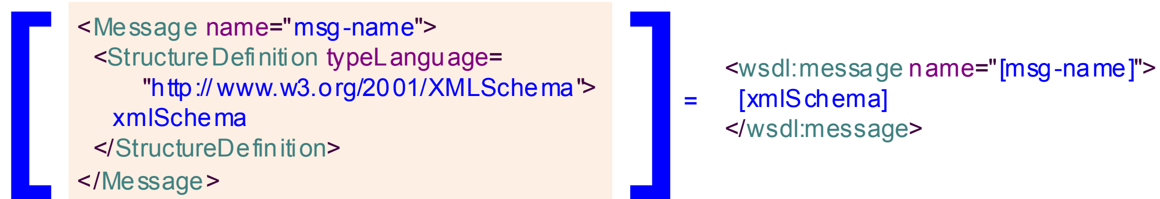
The following figure shows the mapping of an **Abstract Task** to WS-BPEL.



Service Package

Message

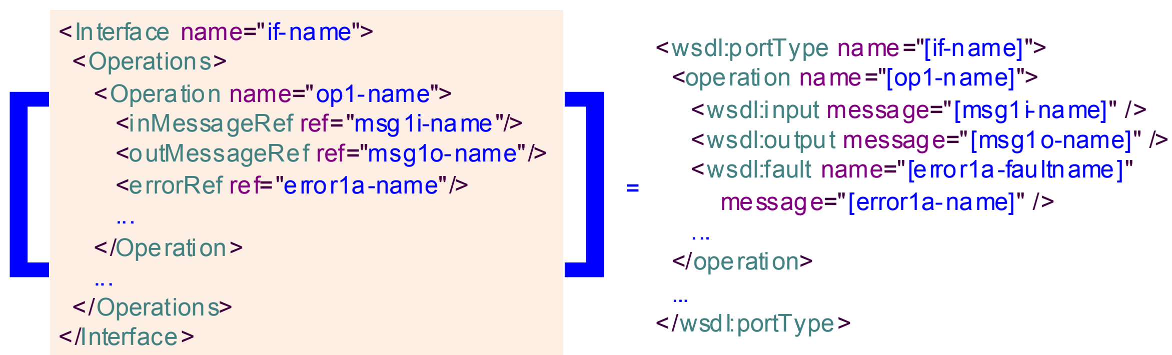
For **Messages** with a scalar data item definition typed by an XML schema definition, the following figure shows the mapping to WS-BPEL, using WSDL 1.1.



The top-level child elements of the XML schema defining the structure of the **BPMN Message** are mapped to the WSDL's message's parts.

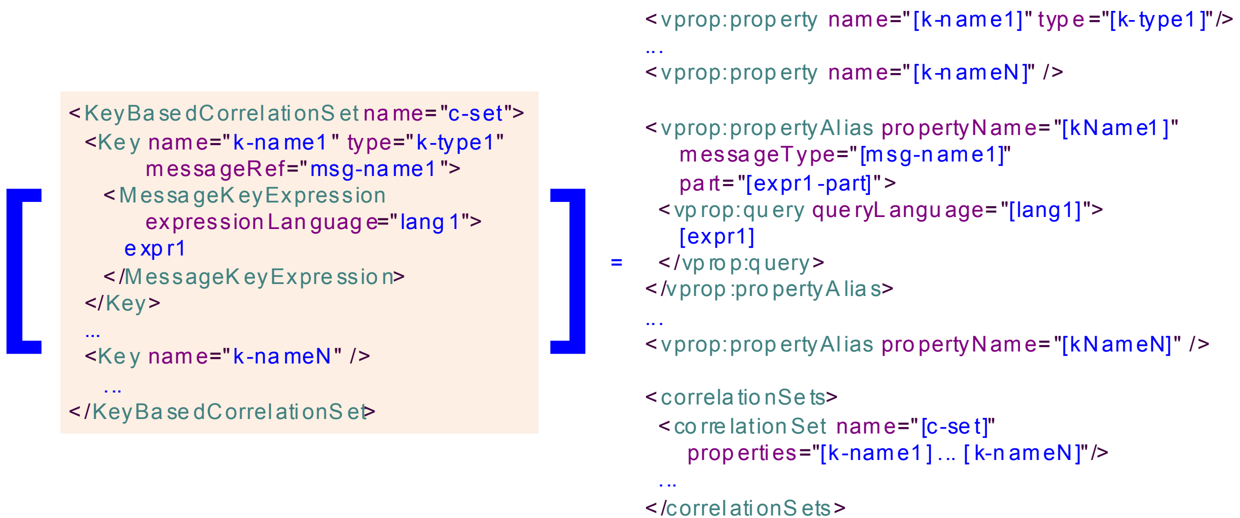
Interface and Operation

The following figure shows the mapping of a **BPMN** interface with its operations to WS-BPEL, using WSDL 1.1.



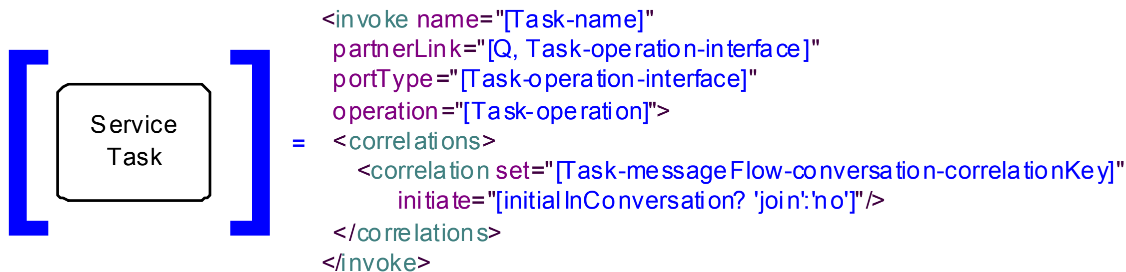
Conversations and Correlation

For those **BPMN** nodes sending or receiving **Messages** (i.e., **Message Events**, **Service**, send or **Receive Tasks**) that have an associated key-based `Correlation Key`, the mapping of that key-based `Correlation Key` is as follows.



The `messageType` of the BPEL property alias is appropriately derived from the `itemDefinition` of the **Message** referenced by the **BPMN Message** key `Expression`. The name of the **Message** part is derived from the **Message** key `Expression`. The **Message** key `Expression` itself is transformed into an `Expression` relative to that part.

The mapping of **Activities** with an associated key-based `Correlation Key` is extended to reference the above BPEL correlation set in the corresponding BPEL `correlations` element. The following figure shows that mapping in the case of a **Service Task** with an associated key-based `Correlation Key`.



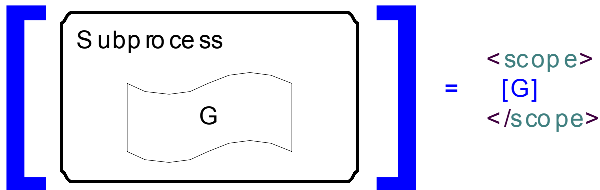
The initiate attribute of the BPEL `correlation` element is set depending on whether or not the associated **Message Flow** initiates the associated **Conversations**, or participates in an already existing **Conversation**. If there are multiple `CorrelationKeys` associated with the **Conversation**, multiple *correlation* elements are used.

Sub-Process Mappings

The following table displays the mapping of an embedded **Sub-Process** with Adhoc="*False*" to a WS-BPEL scope. (This extends the mappings that are defined for all **Activities**--see page 448).

The following figure shows the mapping of a **BPMN Sub-Process** without an **Event Sub-Process**.

The following figure shows the mapping of a **BPMN Sub-Process** with an **Event Sub-Process**. (**Event Sub-Processes** could also be added to a top-level **Process**, in which case their mapping extends correspondingly.)

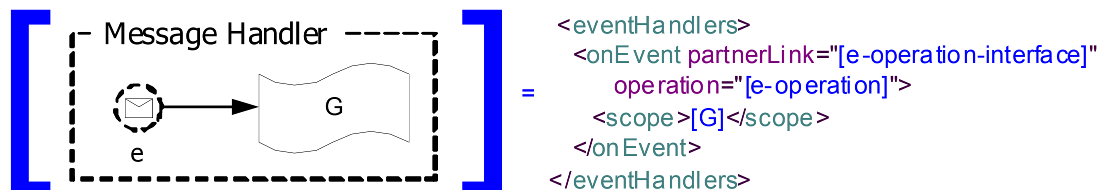


Note that in case of multiple **Event Sub-Processes**, there would be multiple WS-BPEL handlers.

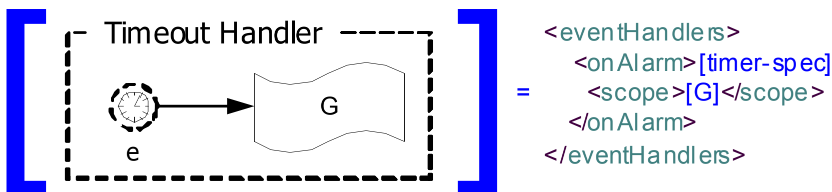
Mapping of Event Sub-Processes

Note that if a **Sub-Process** contains multiple **Event Sub-Processes**, all become handlers of the associated WS-BPEL `scope`, ordered and grouped as specified by WS-BPEL.

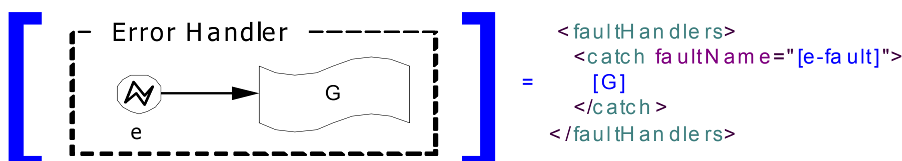
Non-interrupting **Message Event Sub-Processes** are mapped to WS-BPEL event handlers as follows.



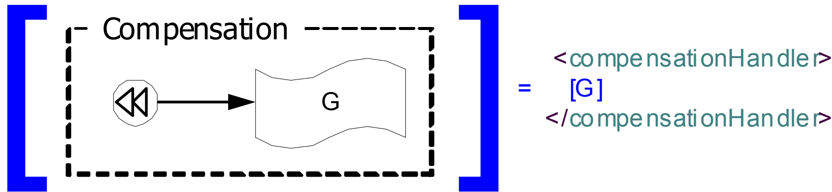
Timer **Event Sub-Processes** are mapped to WS-BPEL event handlers as follows.



Error **Event Sub-Processes** are mapped to WS-BPEL fault handlers as follows.



A **Compensation Event Sub-Process** is mapped to a WS-BPEL compensation handler as follows.



Activity Loop Mapping

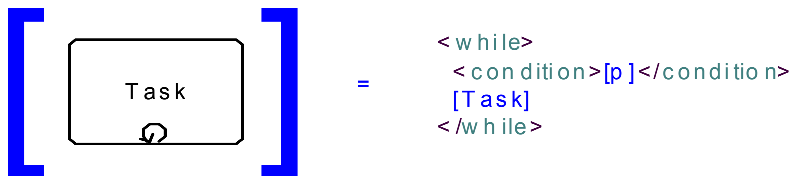
Standard *loops* with a testTime attribute "Before" or "After" execution of the **Activity** map to WS-BPEL `while` and `repeatUntil` activities in a straight-forward manner. When the `LoopMaximum` attribute is used, additional activities are used to maintain a *loop* counter.

Multi-instance **Activities** map to WS-BPEL `forEach` activities in a straight-forward manner.

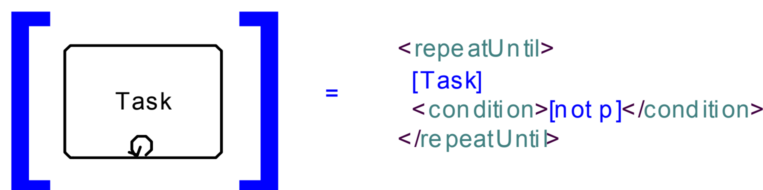
Standard Loops

The mappings for standard *loops* to WS-BPEL are described in the following.

A standard *loop* with testTime= "Before" maps to WS-BPEL as follows, where *p* denotes the *loop* condition.



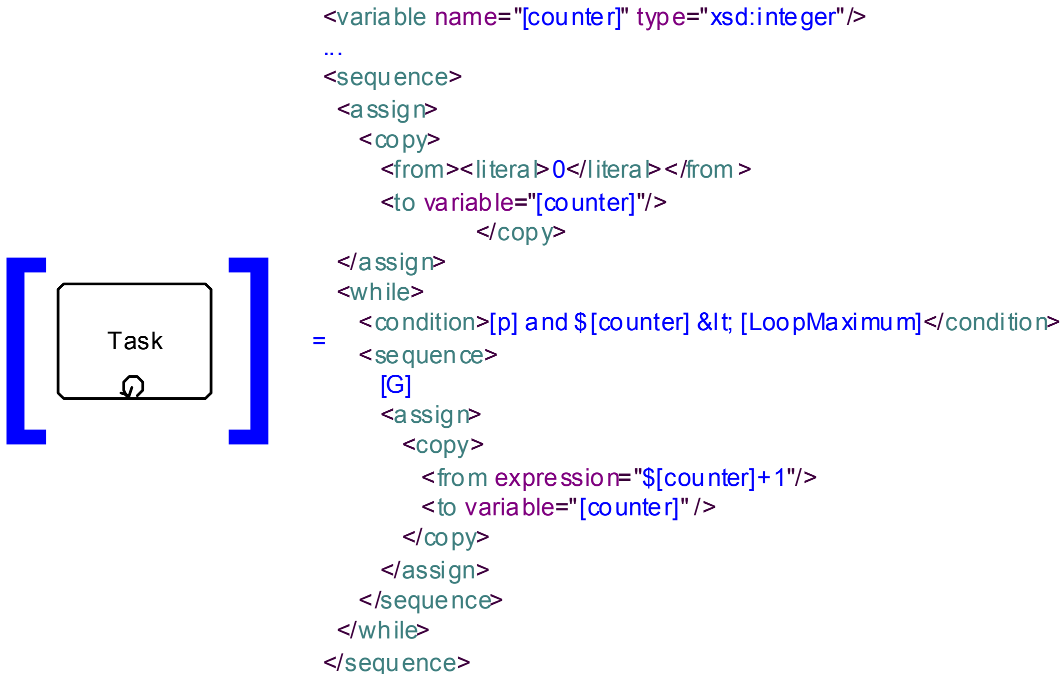
A standard *loop* with testTime= "After" maps as follows, where *p* denotes the *loop* condition.



Dealing with LoopMaximum

When the `LoopMaximum` attribute is specified for an **Activity**, the *loop* requires additional set up for maintaining a counter.

A standard *loop* with testTime="Before" and a `LoopMaximum` attribute maps to WS-BPEL as follows (again, *p* denotes the `loopCondition`).



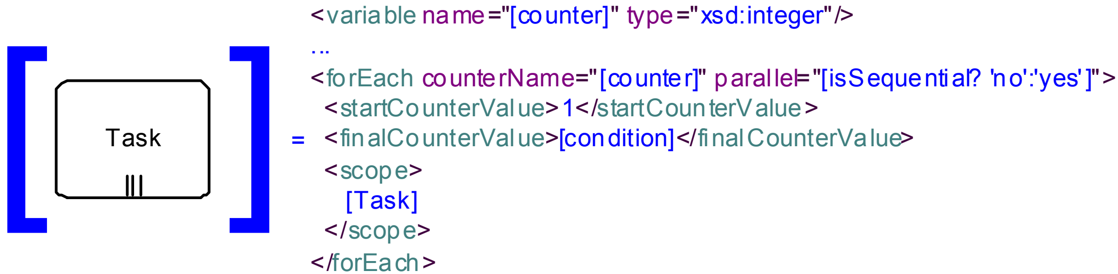
(The notation `[counter]` denotes the unique name of a variable used to hold the counter value; the actual name is immaterial.)

A standard *loop* with testTime="After" and a `LoopMaximum` attribute maps as follows:

(The notation `[counter]` denotes the unique name of a variable used to hold the counter value; the actual name is immaterial.)

Multi-Instance Activities

A **BPMN Multi-Instance Task** with a `multiInstanceFlowCondition` of "All" is mapped to WS-BPEL as follows.



(The notation `[counter]` denotes the unique name of a variable used to hold the counter value; the actual name is immaterial.)

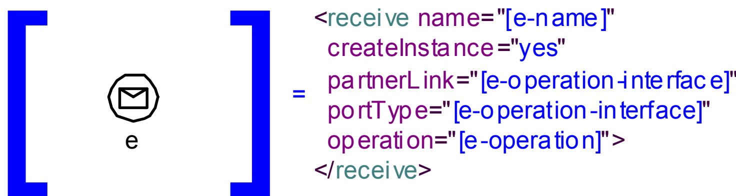
14.2.3 Events

Start Event Mappings

The following sub clauses detail the mapping of **Start Events** to WS-BPEL.

Message Start Events

A **Message Start Event** is mapped to WS-BPEL as shown in the following figure.



The partner link associated with the WS-BPEL receive is derived from the interface referenced by the operation of the **Message Start Event**.

Error Start Events

An **Error Start Event** can only occur in **Event Sub-Processes**. This mapping is described on page 455.

Compensation Start Events

A **Compensation Start Event** can only occur in **Event Sub-Processes**. This mapping is described on page 455.

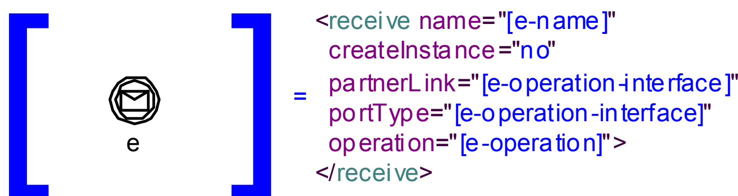
Intermediate Event Mappings (Non-boundary)

The following sub clauses detail the mapping of intermediate non-boundary **Events** to WS-BPEL.

Message Intermediate Events (Non-boundary)

A **Message Intermediate Event** can either be used in normal control flow, similar to a **Send** or **Receive Task** (for *throw* or *catch* **Message Intermediate Events**, respectively), or it can be used in an **Event Gateway**. The latter is described in more detail in “Gateways and Sequence Flows” on page 461.

The following figure describes the mapping of **Message Intermediate Events** to WS-BPEL.

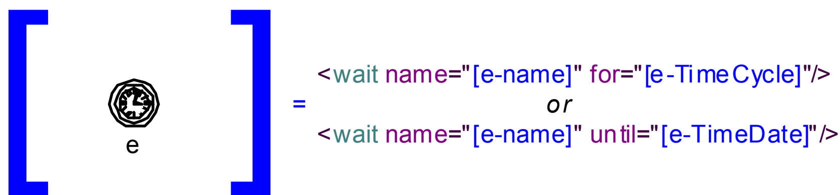


The partner link associated with the WS-BPEL receive is derived from the interface referenced by the operation of the **Message Intermediate Event**.

Timer Intermediate Events (Non-boundary)

A **Timer Intermediate Event** can either be used in normal control flow, or it can be used in an **Event Gateway**. The latter is described in more detail in “Gateways and Sequence Flows” on page 461.

The following figure describes the mapping of a **Timer Intermediate Event** to WS-BPEL – note that one o the mappings shown is chosen depending on whether the **Timer Event’s** `TimeCycle` or `TimeDate` attribute is used.



Compensation Intermediate Events (Non-boundary)

A **Compensation Intermediate Event** with its `waitForCompletion` property set to *true*, that is used within an **Event Sub-Process** triggered through an *error* or through *compensation*, is mapped to WS-BPEL as follows.

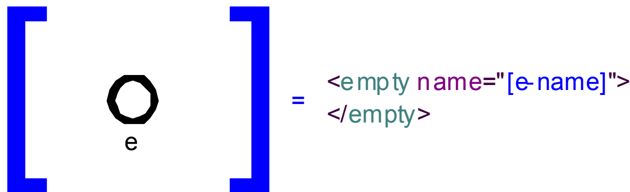
The first mapping is used if the **Compensation Event** does not reference an **Activity**, the second mapping is used otherwise.

End Event Mappings

The following sub clauses detail the mapping of **End Events** to WS-BPEL.

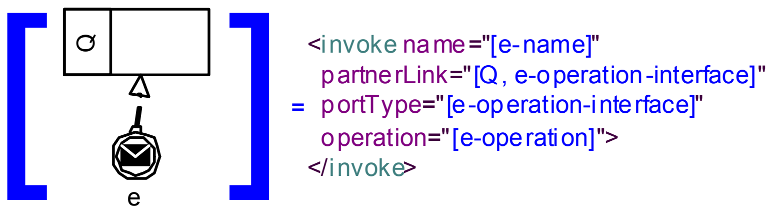
None End Events

A "none" **End Event** marking the end of a **Process** is mapped to WS-BPEL as shown in the following figure.



Message End Events

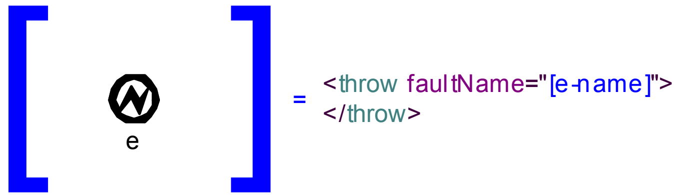
A **Message Start Event** is mapped to WS-BPEL as shown in the following figure.



The partner link associated with the WS-BPEL invoke is derived from both the participant Q that the **Message Intermediate Event** is connected to by a **Message Flow**, and from the `interface` referenced by the `operation` of the **Message Intermediate Event**.

Error End Events

An **Error End Event** is mapped to WS-BPEL as shown in the following figure.



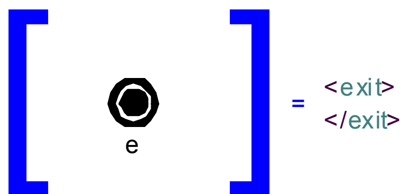
Compensation End Events

A **Compensation End Event** with its `waitForCompletion` property set to *true*, that is used within an **Event Sub-Process** triggered through an *error* or through *compensation*, is mapped to WS-BPEL as follows.

The first mapping is used if the **Compensation Event** does not reference an **Activity**, the second mapping is used otherwise.

Terminate End Events

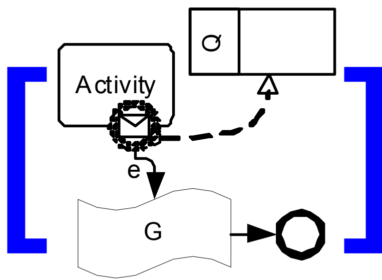
A **Terminate End Event** is mapped to WS-BPEL as shown in the following figure.



Boundary Intermediate Events

Message Boundary Events

A **BPMN Activity** with a non-interrupting **Message** boundary **Event** is mapped to a WS-BPEL scope with an event handler as follows.



=

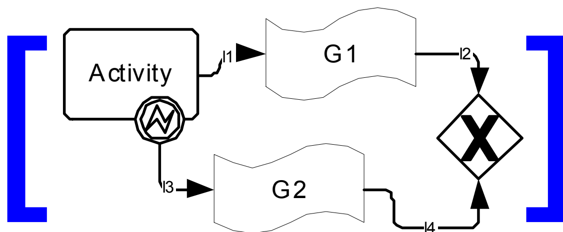
```
<scope>
  <eventHandlers>
    <onEvent partnerLink="[Q, e-operation-interface]"
         operation="[e-operation]">
      <scope>[G]</scope>
    </onEvent>
  </eventHandlers>
  [Activity]
</scope>
```

The partner link associated with the WS-BPEL onEvent is derived from the interface referenced by the operation of the boundary **Message Event**.

The same mapping applies to a non-interrupting boundary **Timer Event**, using a WS-BPEL onAlarm handler instead.

Error Boundary Events

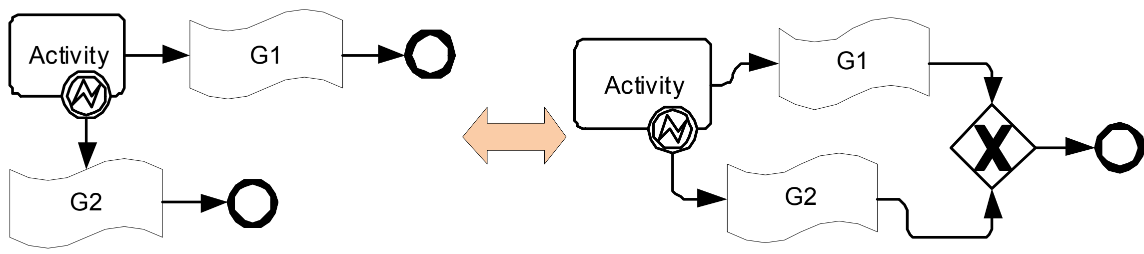
A **BPMN Activity** with a boundary **Error Event** according to the following pattern is mapped as shown.



=

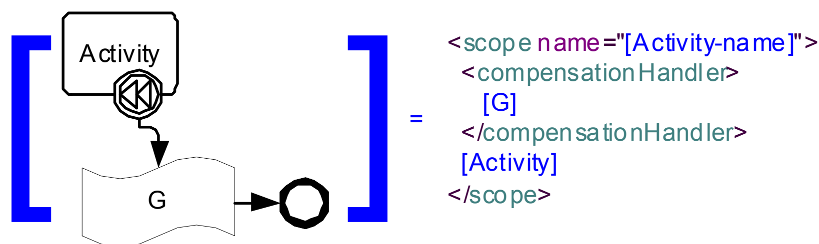
```
<flow>
  <links>
    <link name="[l1]"/>
    ...
    <link name="[l4]"/>
  </links>
  <scope>
    <sources><source linkName="[l1]"/></sources>
    <faultHandlers>
      <catch faultName="[e-error]">
        <empty>
          <sources><source linkName="[l3]"/></sources>
        </empty>
      </catch>
    </faultHandlers>
    [Activity]
  </scope>
  <flow>
    <targets><target linkName="[l1]"/></targets>
    <sources><source linkName="[l2]"/></sources>
    [G1]
  </flow>
  <flow>
    <targets><target linkName="[l3]"/></targets>
    <sources><source linkName="[l4]"/></sources>
    [G2]
  </flow>
  <empty>
    <sources><source linkName="[l2]"/>
    <source linkName="[l4]"/></sources>
  </empty>
</flow>
```

Note that the case where the error handling path doesn't join the main control flow again, is still mapped using this pattern, by applying the following model equivalence.



Compensation Boundary Events

A **BPMN Activity** with a *boundary* **Compensation Event** is similarly mapped as shown.

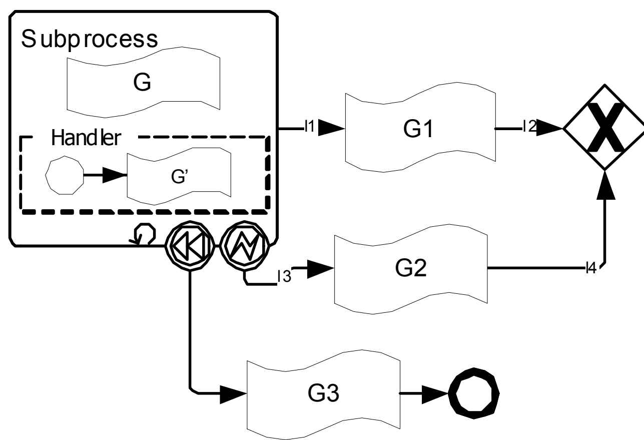


Multiple Boundary Events, and Boundary Events with Loops

If there are multiple boundary **Events** for an **Activity**, their WS-BPEL mappings are super-imposed on the single WS-BPEL `scope` wrapping the mapping of the **Activity**.

When the **Activity** is a standard *loop* or a *multi-instance* and has one or more boundary **Events**, the WS-BPEL *loop* resulting from mapping the **BPMN** *loop* is nested inside the WS-BPEL scope resulting from mapping the **BPMN** boundary **Events**.

The following example shows that mapping for a **Sub-Process** with a nested **Event Sub-Process** that has a standard *loop* with `TestTime`="Before," a boundary **Error Intermediate Event**, and a boundary **Compensation Intermediate Event**.



=

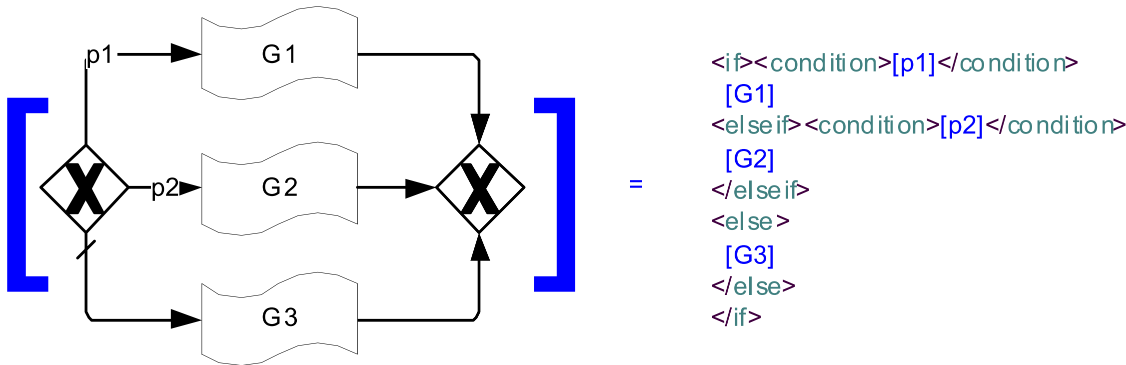
```
<flow>
  <links>
    <link name="[l1]"/>
    ...
    <link name="[l4]"/>
  </links>
  <scope>
    <sources><source linkName="[l1]"/></sources>
    <faultHandlers>
      <catch faultName="[e-error]">
        <empty>
          <sources><source linkName="[l3]"/></sources>
        </empty>
      </catch>
    </faultHandlers>
    <compensationHandler>
      [G3]
    </compensationHandler>
    <while>
      <condition>[p]</condition>
      <scope>
        [Handler]
        [G]
      </scope>
    </while>
  </scope>
  <flow>
    <targets><target linkName="[l1]"/></targets>
    <sources><source linkName="[l2]"/></sources>
    [G1]
  </flow>
  <flow>
    <targets><target linkName="[l3]"/></targets>
    <sources><source linkName="[l4]"/></sources>
    [G2]
  </flow>
  <empty>
    <sources><source linkName="[l2]"/>
    <source linkName="[l4]"/></sources>
  </empty>
</flow>
```

14.2.4 Gateways and Sequence Flows

The mapping of **BPMN Gateways** and **Sequence Flows** is described using **BPMN** blocks following particular patterns.

Exclusive (Data-based) Decision Pattern

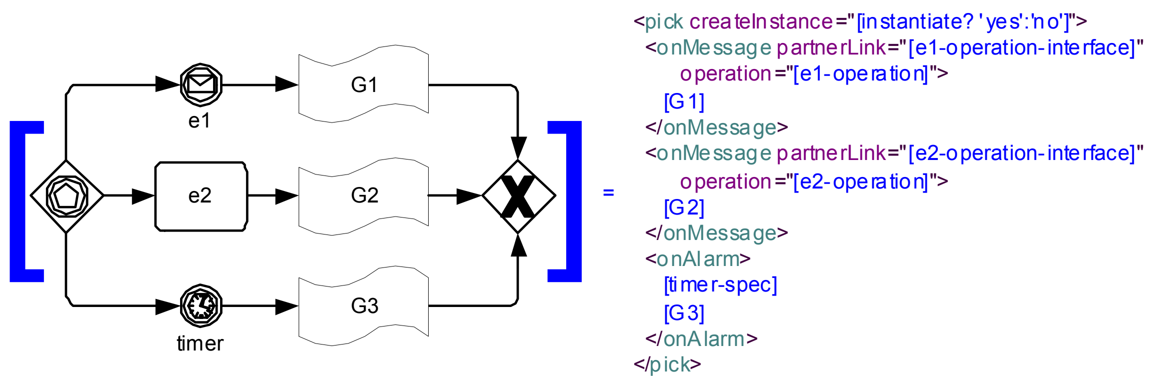
An exclusive data-based decision is mapped as follows.



While this figure shows three branches, the pattern is generalized to *n* branches in an obvious manner.

Exclusive (Event-based) Decision Pattern

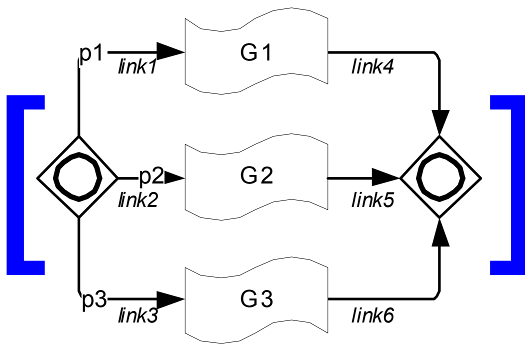
An **Event Gateway** is mapped as follows.



While this figure shows three branches with one **Message Intermediate Event**, one **Receive Task** and one **Timer Intermediate Event**, the pattern is generalized to *n* branches with any combination of the former in an obvious manner. The handling of *Participants* (BPEL partnerLinks), **Event** (operation) and timer details is as specified for **Message Intermediate Events**, **Receive Tasks**, and **Timer Intermediate Events**, respectively. The data flow and associated variables (not shown) are handled as for **Receive Tasks/Message Intermediate Events**.

Inclusive Decision Pattern

An inclusive decision pattern without an otherwise gate is mapped as follows:



=

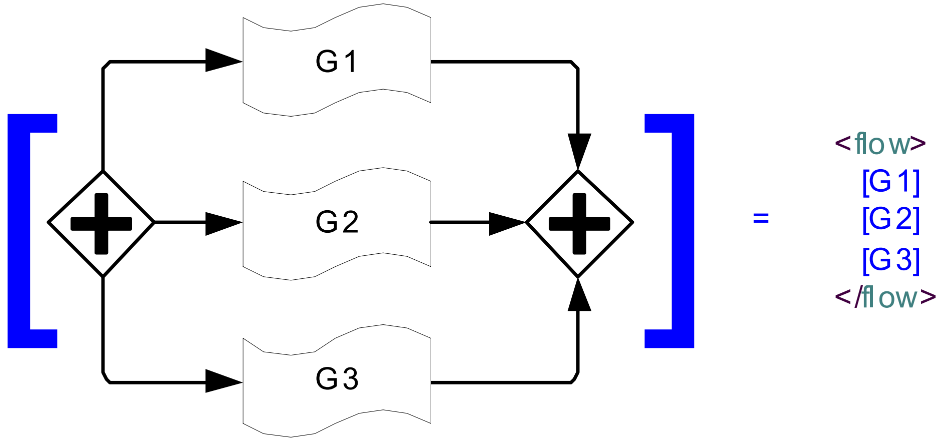
```
<flow>
  <links>
    <link name="[link1]"/>
    ...
    <link name="[link6]"/>
  </links>

  <empty>
    <sources>
      <source linkName="[link1]">
        <transitionCondition>[p1])</transitionCondition>
      </source>
      <source linkName="[link2]">
        <transitionCondition>[p2])</transitionCondition>
      </source>
      <source linkName="[link3]">
        <transitionCondition>[p3])</transitionCondition>
      </source>
    </sources>
  </empty>

  <flow>
    <targets><target linkName="[link1]"/></targets>
    <sources><source linkName="[link4]"/></sources>
    [G1]
  </flow>

  <flow>
    <targets><target linkName="[link2]"/></targets>
    <sources><source linkName="[link5]"/></sources>
    [G2]
  </flow>

  <flow>
    <targets><target linkName="[link3]"/></targets>
    <sources><source linkName="[link6]"/></sources>
    [G3]
  </flow>

  <empty>
    <targets>
      <target linkName="[link4]"/>
      <target linkName="[link5]"/>
      <target linkName="[link6]"/>
    </targets>
  </empty>
</flow>
```

While this figure shows three branches, the pattern is generalized to *n* branches in an obvious manner.

Note that link names in WS-BPEL MUST follow the rules of an XML NCName. Thus, the mapping of the **BPMN Sequence Flow** name attribute MUST appropriately canonicalize that name, possibly ensuring uniqueness, e.g., by appending a unique suffix. This is capture by the [linkName] notation.

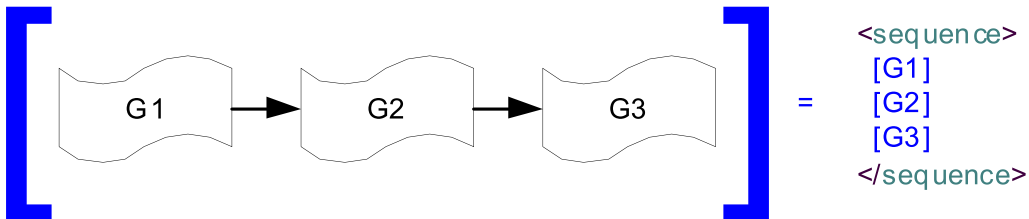
Parallel Pattern

A parallel fork-join pattern is mapped as follows.



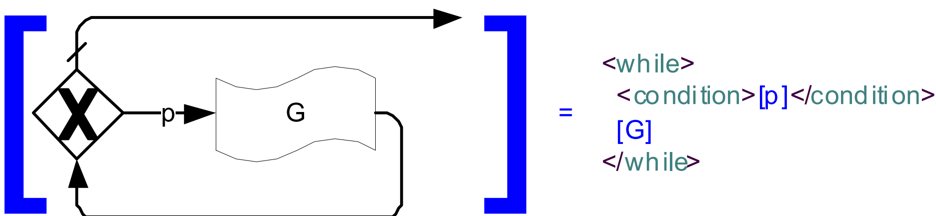
Sequence Pattern

A **BPMN** block consisting of a series of **Activities** connected via (unconditional) **Sequence Flows** is mapped to a WS-BPEL `sequence`:

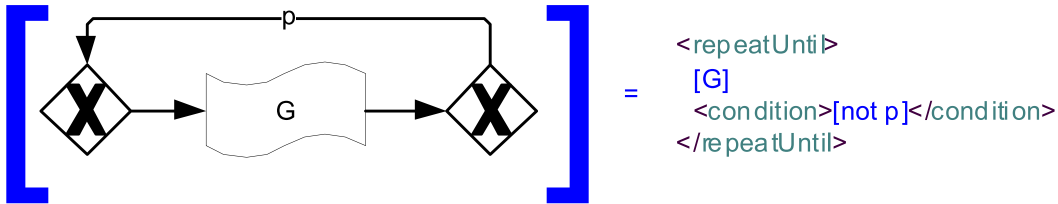


Structured Loop Patterns

A **BPMN** block consisting of a structured *loop* of the following pattern is mapped to a WS-BPEL `while`.



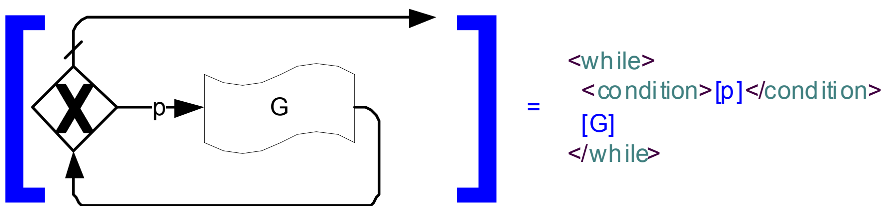
A **BPMN** block consisting of a structured *loop* of the following pattern is mapped to a WS-BPEL `repeatUntil`.



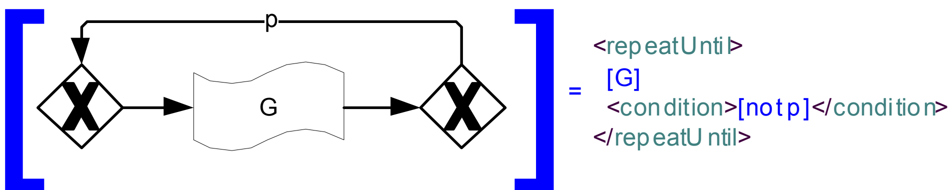
Handling Loops in Sequence Flows

Loops are created when the flow of the **Process** moves from a downstream object to an upstream object. There are two types of *loops* that are WS-BPEL mappable: while *loops* and repeat *loops*.

A while *loop* has the following structure in **BPMN** and is mapped as shown.



A repeat *loop* has the following structure in **BPMN** and is mapped as shown.

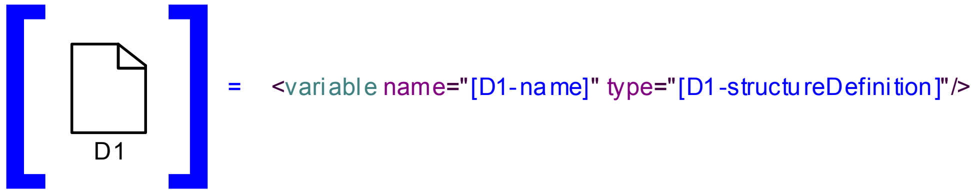


14.2.5 Handling Data

Data Objects

BPMN Data Objects are mapped to WS-BPEL variables. The `itemDefinition` of the **Data Object** determines the XSD type of that variable.

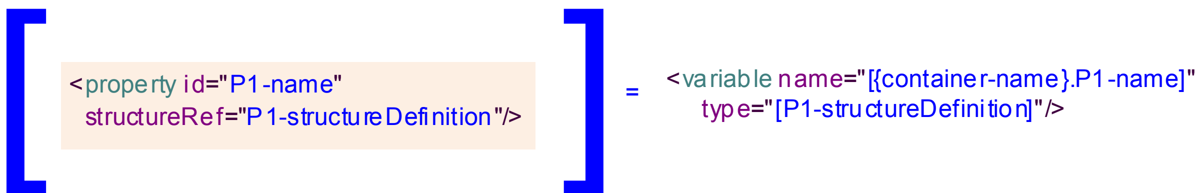
Data Objects occur in the context of a **Process** or **Sub-Process**. For the associated WS-BPEL `process` or WS-BPEL `scope`, a variable is added for each **Data Object** in the corresponding WS-BPEL `variables` sub clause, as follows:



Properties

BPMN properties can be contained in a **Process**, **Activity**, or an **Event**, here named the “container” of the property. A **BPMN** property is mapped to a WS-BPEL variable. Its name is derived from the name of its container and the name of the property. Note that in the case of different containers with the same name and a contained property of the same name, the mapping to WS-BPEL ensures the names of the associated WS-BPEL variables are unique. The `itemDefinition` of the property determines the XSD type of that variable.

A **BPMN Process** property is mapped to a WS-BPEL global variable. A **BPMN Event** `property` is mapped to a WS-BPEL variable contained in the WS-BPEL scope representing the immediately enclosing **Sub-Process** of the **Event** (or a global variable in case the **Event** is an immediate child of the **Process**). For a **BPMN Activity** property, two cases are distinguished: In case of a **Sub-Process**, the WS-BPEL variable is contained in the WS-BPEL scope representing the **Sub-Process**. For all other **BPMN Activity** properties, the WS-BPEL variable is contained in the WS-BPEL scope representing the immediately enclosing **Sub-Process** of the **Activity** (or a global variable in case the **Activity** is an immediate child of the **Process**).



Input and Output Sets

For a **Send Task** and a **Service Task**, the single input set is mapped to a WSDL message defining the input of the associated WS-BPEL activity. The inputs map to the message parts of the WSDL message. For a **Receive Task** and a **Service Task**, the single output set is mapped to a WSDL message defining the output of the associated WS-BPEL activity. The outputs map to the message parts of the WSDL message.

The structure of the WSDL message is defined by the `itemDefinitions` of the data inputs of the input set.

```
[ <inputSet name="iset">
    <dataInput name="input1">
      <structureDefinition structure="type1"/>
    </dataInput>
    ...
  </inputSet> ]
=
<wsdl:message name="[iset-name]">
  <part name="[input1-name]" type="[type1]"/>
  ...
</wsdl:message>
```

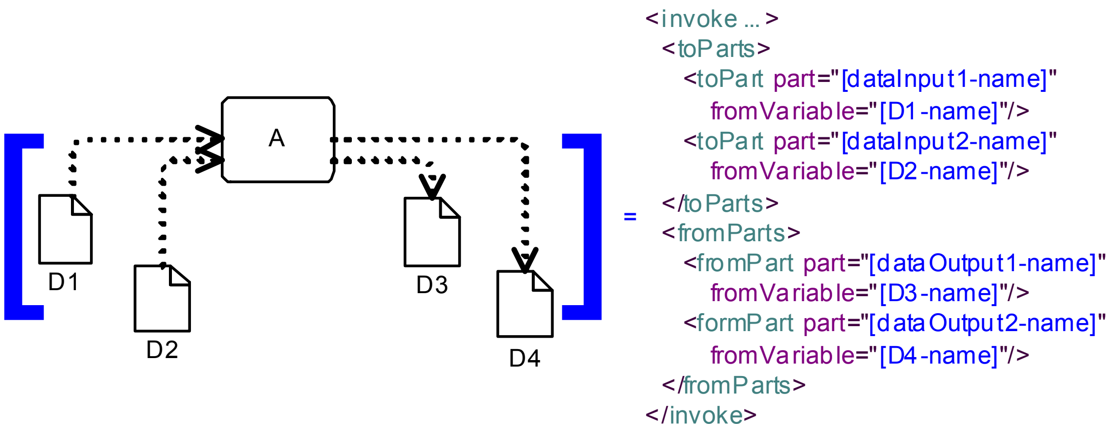
For the data outputs of the output set, the WSDL message looks as follows.

```
[ <outputSet name="oset">
    <dataOutput name="output1">
      <structureDefinition structure="type3"/>
    </dataOutput>
    ...
  </outputSet> ]
=
<wsdl:message name="[oset-name]">
  <part name="[output1-name]" type="[type3]"/>
  ...
</wsdl:message>
```

Data Associations

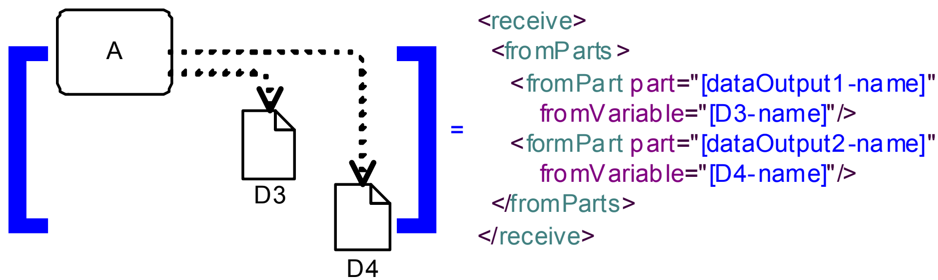
In this sub clause, we assume that the input set of the **Service Task** has the same structure as its referenced input **Message**, and the output set of the **Service Task** has the same structure as its reference output **Message**. If this is not the case, assignments are needed, and the mapping is as described in the next sub clause.

Data associations to and from a **Service Task** are mapped as follows.

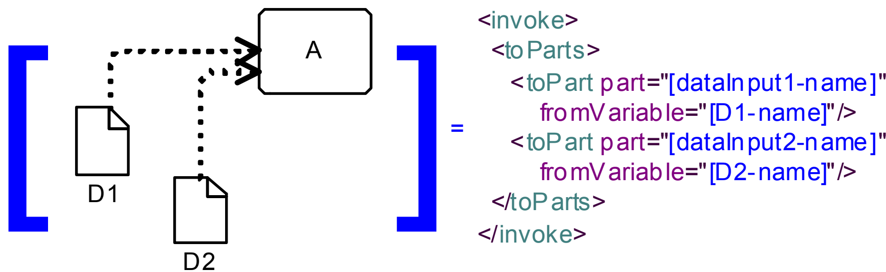



```
<invoke ... >
  <toParts>
    <toPart part="[dataInput1-name]"
      fromVariable="[D1-name]"/>
    <toPart part="[dataInput2-name]"
      fromVariable="[D2-name]"/>
  </toParts>
  <fromParts>
    <fromPart part="[dataOutput1-name]"
      fromVariable="[D3-name]"/>
    <formPart part="[dataOutput2-name]"
      fromVariable="[D4-name]"/>
  </fromParts>
</invoke>
```

Data associations from a **Receive Task** are mapped as follows.



Data associations to a **Send Task** are mapped as follows.



Expressions

BPMN `Expressions` specified using XPath (e.g., a condition `Expression` of a **Sequence Flow**, or a timer cycle `Expression` of a **Timer Intermediate Event**) are used as specified in **BPMN**, rewriting access to **BPMN** context to refer to the mapped BPEL context.

The **BPMN** XPath functions for accessing context from the perspective of the current **Process** are mapped to BPEL XPath functions for context access as shown in the following table. This is possible because the arguments MUST be literal strings.

Table 14.2 – Expressions mapping to WS-BPEL

BPMN context access	BPEL context access
getDataobject(dataObjectName)	$[dataObjectName]
getProcessProperty(propertyName)	$[{processName}.propertyName] where the right processName is statistically derived.
getActivityProperty(activityName, propertyName)	$[activityName.propertyName]
getEventProperty(eventName, propertyName)	$[eventName.propertyName]

Assignments

For a **Service Task** with assignments, the WS-BPEL mapping results in a sequence of an assign activity, an invoke activity and another assign activity. The first assign deals with creating the service request **Message** from the data inputs of the **Task**, the second assign deals with creating the data outputs of the **Task** from the service response **Message**.

14.3 Extended BPMN-BPEL Mapping

Additional sound **BPMN Process** models whose block hierarchy contains blocks that have not been addressed in the previous sub clause can be mapped to WS-BPEL. For such **BPMN Process** models, in many cases there is no preferred single mapping of a particular block, but rather, multiple WS-BPEL patterns are possible to map that block to. Also, additional **BPMN** constructs can be mapped by using capabilities not available at the time of producing this specification, such as the upcoming OASIS BPEL4People standard to map **BPMN User Tasks**, or other WS-BPEL extensions.

Rather than describing or even mandating the mapping of such **BPMN** blocks, this specification allows for a semantic mapping of a **BPMN Process** model to an executable WS-BPEL process: The observable behavior of the target WS-BPEL process MUST match the operational semantics of the mapped **BPMN Process**. Also, the mappings described in sub clause 15.1 SHOULD be used where applicable.

14.3.1 End Events

End Events can be combined with other **BPMN** objects to complete the merging or joining of the paths of a WSBPEL structured element (see Figure 7.3).

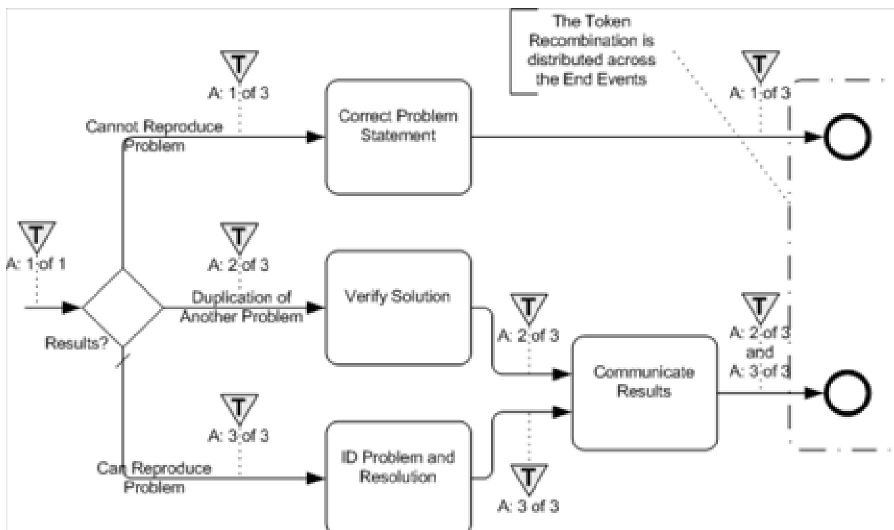


Figure 14.2 – An example of distributed *token* recombination

14.3.2 Loop/Switch Combinations From a Gateway

This type of *loop* is created by a **Gateway** that has three or more *outgoing* **Sequence Flows**. One **Sequence Flow** *loops* back *upstream* while the others continue *downstream* (see Figure 14.3). Note that there might be intervening **Activities** prior to when the **Sequence Flow** *loops* back *upstream*.

- This maps to both a WSBPEL `while` and a `switch`. Both activities will be placed within a *sequence*, with the `while` preceding the `switch`.
- For the `while`:
  - The *Condition* for the **Sequence Flow** that *loops* back *upstream* will map to the *condition* of the *while*.
  - All the **Activities** that span the distance between where the *loop* starts and where it ends, will be mapped and placed within the **Activity** for the *while*, usually within a *sequence*.
- For the `switch`:
  - For each additional *outgoing* **Sequence Flows** there will be a *case* for the *switch*.

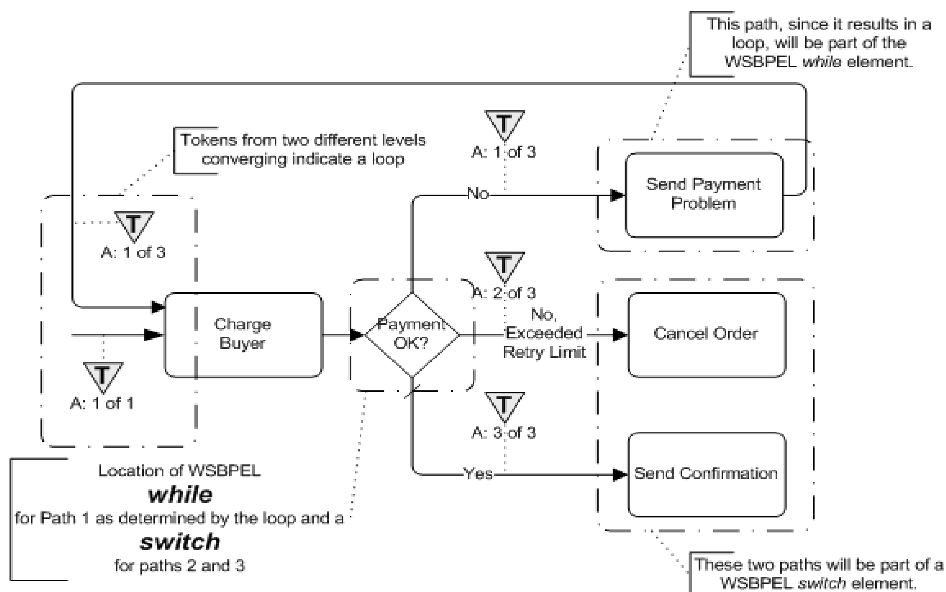


Figure 14.3 – An example of a loop from a decision with more than two alternative paths

14.3.3 Interleaved Loops

This is a situation where there are at least two *loops* involved and they are not nested (see Figure 14.4). Multiple looping situations can map, as described above, if they are in a sequence or are fully nested (e.g., one `while` inside another `while`). However, if the *loops* overlap in a non-nested fashion, as shown in the figure, then the structured element `while` cannot be used to handle the situation. Also, since a `flow` is acyclic, it cannot handle the behavior either.

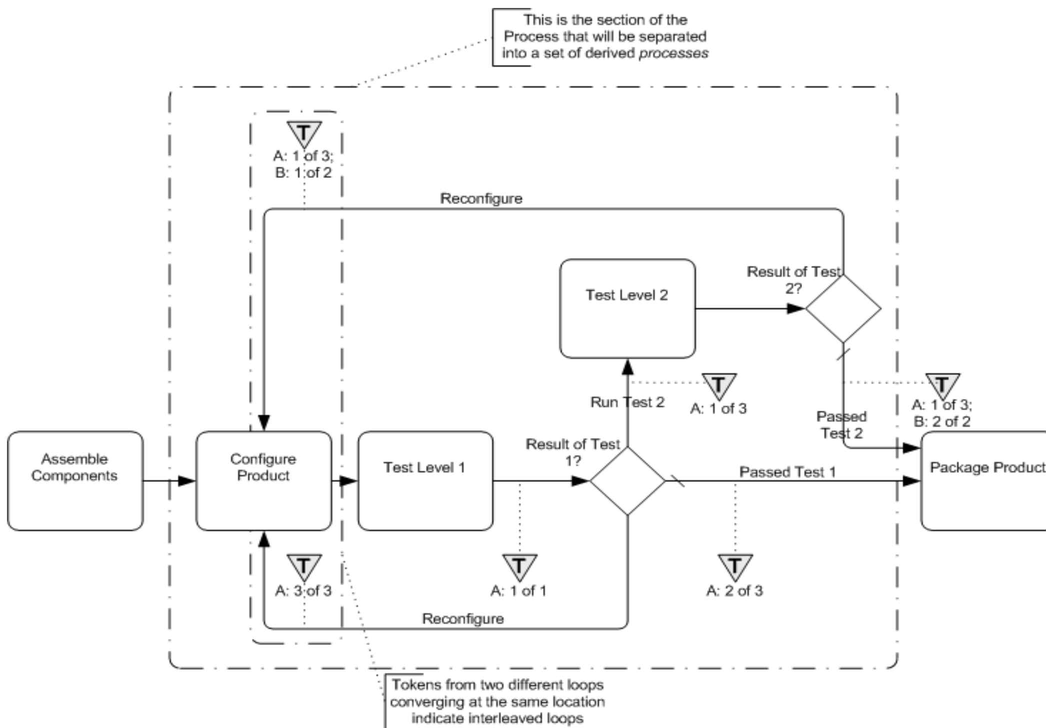


Figure 14.4 – An example of interleaved loops

To handle this type of behavior, parts of the WSBPEL `process` will have to be separated into one or more derived `processes` that are spawned from the main `process` and will also spawn or call each other (note that the examples below are using a spawning technique). Through this mechanism, the linear and structured elements of WSBPEL can provide the same behavior that is shown through a set of cycles in a single **BPMN** diagram. To do this:

- The looping section of the **Process**, where the *loops* first merge back (*upstream*) into the flow until all the paths have merged back to *Normal Flow*, SHALL be separated from the main WSBPEL `process` into a set of derived `processes` that will spawn each other until all the looping conditions are satisfied.
- The section of the `process` that is removed will be replaced by a (one-way) `invoke` to spawn the derived `process`, followed by a `receive` to accept the *message* that the looping sections have completed and the main `process` can continue (see Figure 14.5).
- The name of the `invoke` will be in the form of:
    - "Spawn_[(loop target)activity.Name]_Derived_Process"
    - The name of the *receive* will be in the form of:
    - [(loop target)activity.Name]_Derived_Process_Completed"

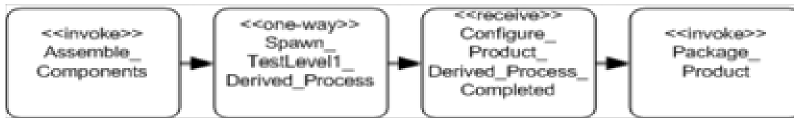


Figure 14.5 – An example of the WSBPEL pattern for substituting for the derived `Process`

For each location in the **Process** where a **Sequence Flow** connects *upstream*, there will be a separate derived WSBPEL `process`.

- The name of the derived `process` will be in the form of:
  - "[(loop target)activity.Name]_Derived_Process"
- All **Gateways** in this sub clause will be mapped to `switch` elements, instead of `while` elements (see Figure below).
- Each time there is a **Sequence Flow** that *loops* back *upstream*, the **Activity** for the `switch case` will be a (one-way) `invoke` that will spawn the appropriate derived `process`, even if the `invoke` spawns the same `process` again.
- The name of the `invoke` will be the same as the one described above.
- At the end of the derived `process` a (one-way) `invoke` will be used to signal the main `process` that all the derived activities have completed and the main `process` can continue.
- The name of the `invoke` will be in the form of:
  - "[(loop target)activity.Name]_Derived_Process_Completed"

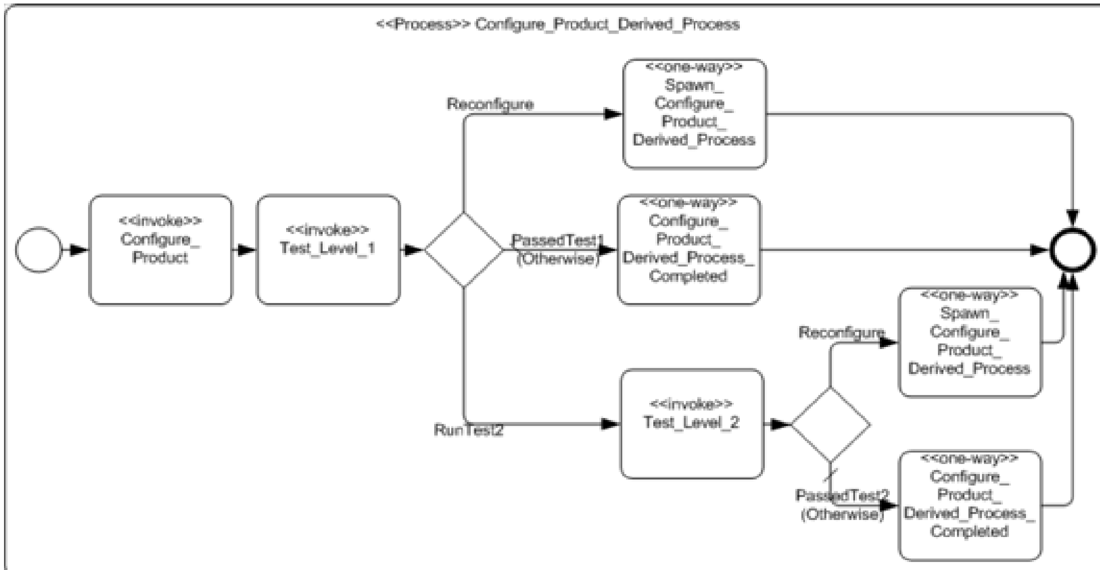


Figure 14.6 – An example of a WSBPEL pattern for the derived `Process`

14.3.4 Infinite Loops

This type of *loop* is created by a **Sequence Flow** that *loops* back without an intervening **Gateway** to create alternative paths (see Figure 14.7). While this can be a modeling error most of the time, there can be situations where this type of *loop* is desired, especially if it is placed within a larger **Activity** that will eventually be interrupted.

- This will map to a `while` activity.
- The condition of the `while` will be set to an `Expression` that will never evaluate to *true*, such as `condition` "1 = 0."
- All the activities that span the distance between where the *loop* starts and where it ends, will be mapped and placed within the activity for the `while`, usually within a `sequence`.

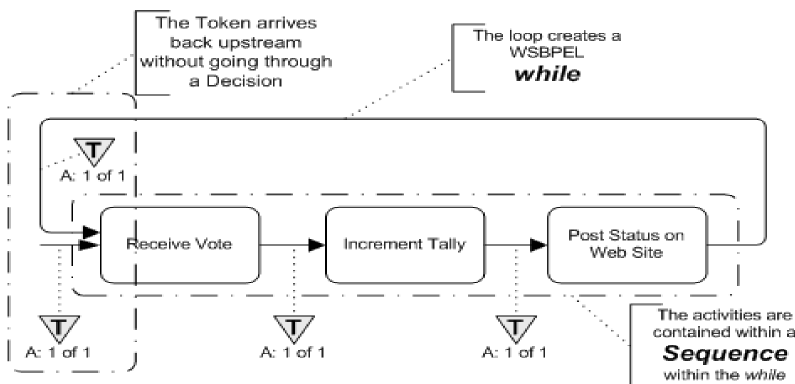


Figure 14.7 – An example: An infinite loop

14.3.5 BPMN Elements that Span Multiple WSBPEL Sub-Elements

Figure 14.8 illustrates how **BPMN** objects can exist in two separate sub-elements of a WSBPEL structured element at the same time. Since **BPMN** allows free form connections of **Activities** and **Sequence Flows**, it is possible that two (or more) **Sequence Flows** will merge before all the **Sequence Flows** that map to a WSBPEL structure element have merged. The sub-elements of a WSBPEL structured elements are also self-contained and there is no cross sub-element flow. For example, the `cases` of a `switch` cannot interact; that is, they cannot share activities. Thus, one **BPMN Activity** will need to appear in two (or more) WSBPEL structured elements. There are two possible mechanisms to deal with the situation:

- First, the activities are simply duplicated in all appropriate WSBPEL elements.
- Second, the activities that need to be duplicated can be removed from the main **Process** and placed in a derived process that is called (*invoked*) from all locations in the WSBPEL elements as needed.
  - The name of the derived process will be in the form of:
    - "[(target)object.Name]_Derived_Process"

Figure 14.8 displays this issue with an example. In that example, two **Sequence Flows** merge into the "Include History of Transactions" **Task**. However, the Decision that precedes the **Task** has three alternatives. Thus, the Decision maps to a WSBPEL `switch` with three `cases`. The three `cases` are not closed until the "Include Standard Text" **Task**, downstream. This means that the "Include History of Transactions" **Task** will actually appear in two of the three *cases* of the `switch`.

Note – the use of a WSBPEL `flow` will be able to handle the behavior without duplicating activities, but a `flow` will not always be available for use in these situations, particularly if a WSBPEL `pick` is requested.

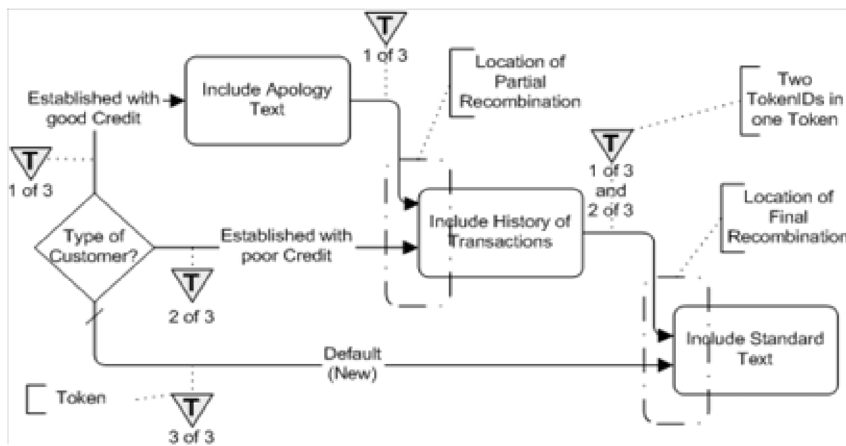


Figure 14.8 – An example: Activity that spans two paths of a WSBPEL structured element

15 Exchange Formats

15.1 Interchanging Incomplete Models

In practice, it is common for models to be interchanged before they are complete. This occurs frequently when doing iterative modeling, where one user (such as a subject matter expert or business person) first defines a high-level model, and then passes it on to another user to be completed and refined.

Such "incomplete" models are ones in which all of the mandatory attributes have not yet been filled in, or the cardinality lowerbound of attributes and associations has not been satisfied.

XMI allows for the interchange of such incomplete models. In **BPMN**, we extend this capability to interchange of XML files based on the **BPMN XSD**. In such XML files, implementers are expected to support this interchange by:

- Disregarding missing attributes that are marked as 'required' in the XSD.
- Reducing the lower bound of elements with 'minOccurs' greater than 0.

15.2 Machine Readable Files

BPMN 2.0.2 machine readable files, including XSD, XMI, and XSLT files can be found in OMG Document dtc/2010-05-04, which is a zip file containing all the files:

- XSD files are found under the XSD folder of the zip file, and the main file is XSD/BPMN20.xsd.
- XMI files are found under the XMI folder of the zip file, and the main file is XSD/BPMN20.cmof.
- XSLT files are found under the XSLT folder of the zip file.

15.3 XSD

15.3.1 Document Structure

A domain-specific set of model elements is interchanged in one or more **BPMN** files. The root element of each file MUST be <bpmn:definitions>. The set of files MUST be self-contained, i.e., all definitions that are used in a file MUST be imported directly or indirectly using the <bpmn:import> element.

Each file MUST declare a "targetNamespace" that MAY differ between multiple files of one model.

BPMN files MAY import non-**BPMN** files (such as XSDs and WSDLs) if the contained elements use external definitions.

Example:

main.bpmn

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn:definitions xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL”
    targetNamespace="sample1.main" xmlns:main="sample1.main" xmlns:s1="sample1.semantic1">
    <bpmn:import location="semantic1.bpmn" namespace="sample1.semantic1"
        importType="http://www.omg.org/spec/BPMN/20100524/MODEL”  />
    <bpmn:import location="diagram1.bpmn" namespace="sample1.diagram1"
```

```
        importType="http://www.omg.org/spec/BPMN/20100524/MODEL" />
    <bpmn:collaboration>
        <bpmn:participant processRef="s1:process1" id="collaboration1"></bpmn:participant>
    <!--more content here -->
    </bpmn:collaboration>
</bpmn:definitions>
```

semantic1.bpmn

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn:definitions xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"
      targetNamespace="sample1.semantic1"
    xmlns:s1="sample1.semantic1">
    <bpmn:process id="process1">
    <!-- content here -->
    </bpmn:process>
</bpmn:definitions>
```

diagram1.bpmn

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn:definitions xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/DI"
    targetNamespace="sample1.diagram1"
    xmlns:bpmndi="http://www.omg.org/spec/BPMNDI/1.0.0"
    xmlns:d1="sample1.diagram1" xmlns:s1="sample1.semantic1"
    xmlns:main="sample1.main">
    <bpmndi:BPMNDiagram scale="1.0" unit="Pixel">
        <bpmndi:BPMNPlane element="main:collaboration1">
        <!-- content here -->
        </bpmndi:BPMNPlane>
    </bpmndi:BPMNDiagram>
</bpmn:definitions>
```

15.3.2 References within the BPMN XSD

All **BPMN** elements contain IDs and within the **BPMN** XSD, references to elements are expressed via these IDs. The XSD IDREF type is the traditional mechanism for referencing by IDs, however it can only reference an element within the same file. The **BPMN** XSD supports referencing by ID, across files, by utilizing QNames. A QName consists of two parts: an optional namespace prefix and a local part. When used to reference a **BPMN** element, the local part is expected to be the ID of the element.

For example, consider the following **Process**

```
<process name="Patient Handling" id="Patient_Handling_Process_ID1"> ... </process>
```

When this **Process** is referenced from another file, the reference would take the following form:

```
processRef="process_ns:Patient_Handling_Process_ID1"
```

where “process_ns” is the namespace prefix associated with the process namespace upon import, and “Patient_Handling_Process_ID1” is the value of the id attribute for the **Process**.

The **BPMN** XSD utilizes IDREFs wherever possible and resorts to QName only when references can span files. In both situations however, the reference is still based on IDs.

15.4 XMI

XMI allows the use of tags to tailor the documents that are produced using XMI. The following tags have been explicitly set for serializing BPMN 2.0 models; the others are left at their default values:

- tag nsURI set to "http://www.omg.org/spec/BPMN/20100524/XMI"
- tag nsPrefix set to “bpmn”

The **BPMN 2.0 XMI** for the interchange of diagram information will be published once the OMG Diagram Definition RFP process has produced a specification that is sufficiently complete such that a future BPMN RFP/FTF/RTF can align the BPMN specification with the Diagram Definition specification.

15.5 XSLT Transformation between XSD and XMI

- The XSLT transformation from XSD to XMI is in the file XSLT/BPMN20-ToXMI.xslt
- The XSLT transformation from XMI to XSD is in the file XSLT/BPMN20-FromXMI.xslt

Annex A
Changes from v1.2

(informative)

A.1 Changes from BPMN, v1.2

There have been notational and technical changes to the BPMN International Standard.

The major notational changes include:

- The addition of a Choreography diagram
- The addition of a Conversation diagram
- Non-interrupting Events for a Process
- Event Sub-Processes for a Process

The major technical changes include:

- A formal metamodel as shown through the class diagram figures
- Interchange formats for abstract syntax model interchange in both XMI and XSD
- Interchange formats for diagram interchange in both XMI and XSD
- XSLT transformations between the XMI and XSD formats

Other technical changes include:

- Reference Tasks are removed. These provided reusability within a single diagram, as compared to GlobalTasks, which are resuable across multiple diagrams. GlobalTasks can be used instead of Reference Tasks, to simplify the language and implementations.

Annex B
Diagram Interchange

(non-normative)

B.1 Scope

This annex provides documentation for a relevant subset of an alpha version of a Diagram Definition (DD) specification that is being referenced by this International Standard (in Clause 13 - BPMN DI). The (complete version of the) DD specification is still going through a separate submission/approval process and once finalized and adopted, a future revision of this specification may replace this annex by a reference to that adopted DD specification.

The Diagram Definition specification provides a basis for modeling and interchanging graphical notations, specifically node and edge style diagrams as found in BPMN, UML and SysML, for example, where the notations are tied to abstract language syntaxes defined with MOF. The specification addresses the requirements in the Diagram Definition RFP (ad/2007-09-02).

B.2 Architecture

The DD architecture distinguishes two kinds of graphical information, depending on whether language users have control over it. Graphics that users have control over, such as position of nodes and line routing points, are captured for interchange between tools. Graphics that users do not have control over, such as shape and line styles defined by language standards are not interchanged because they are the same in all diagrams conforming to the language. The DD architecture has two models to enable specification of these two kinds of graphical information, Diagram Interchange (DI) and Diagram Graphics (DG).(both models share common elements from a Diagram Common (DC) model). The DI and DG models are shown in Figure B.1 by bold outlined boxes on the left and right, respectively.

The DD architecture expects language specification to define mappings between interchanged and non-interchanged graphical information, but does not restrict how it is done. This is shown in Figure B.1 by a shaded box labeled "CS Mapping Specification" in the middle section. The DD specification gives examples of mappings in QVT, but does not define or recommend any particular mapping language. The overall architecture resembles typical model-view-controllers, which separate visual rendering from underlying models, and provide a way to keep visuals and models consistent.

The first few steps of using the DD architecture are:

1. An abstract language syntax is defined separately from DD by instantiating MOF (abstract syntaxes are sometimes called "metamodels"). This is shown in Figure B.1 by a shaded box labeled "AS" at the far middle left (the "M" levels in the figure are described in the UML 2 Infrastructure (formal/2009-02-04)).

2. Language users model their applications by instantiating elements of abstract syntax, usually through tooling for the language. This is shown in Figure B.1 by the dashed arrow on the far lower left linked to a box labeled "Model."

3. Users typically see graphical depictions of their models in tools. This is shown in Figure B.1 by a box on the lower right labeled "Graphics."

Users expect their graphics to appear again in other tools after models are interchanged. The DD architecture enables this in two parts, one for graphical information that is interchanged, and another for graphical information that is not. The interchanged information is captured in the next few steps:

4. The portion of graphics that users have control over is captured for interchange, such as node position and line routing points. This is shown in Figure B.1 by a box labeled “Diagram” on the lower left. This information is linked to user models (instances of abstract syntax), as shown by the arrow to the Model box.

5. User diagram interchange information is instantiated from a model defined along with the abstract syntax. This is shown in Figure B.1 by a shaded box labeled “AS DI” on the left. Elements of this model are linked to elements of abstract syntax, specifying which kinds of diagram interchange information depict which kinds of user model elements. Diagram interchange models would typically be defined by the same community that defines the abstract syntax, as part of the overall language specification.

6. Elements of language-specific diagram interchange models (AS DI) specialize elements of the Diagram Interchange (DI), which is a model provided by this specification for typically needed diagram interchange information, as node position and line routing points. This is shown in Figure B.1 by the bold box labeled “DI” on the left, with specialization shown with a hollow headed arrow (specialization here is MOF generalization and property subsetting and redefinition, or XSD subclassing, where DI has the general elements, and AS DI has the specific elements). DI elements cannot be instantiated to capture diagram interchange information by themselves, they are almost entirely abstract. This specification provides normative CMOF and XSD artifacts for DI.

The final part of using the DD architecture captures graphical information that is not interchanged:

7. Language specifications specify mappings from their diagram interchange models (instances of AS DI) to instances of Diagram Graphics (DG), which is a model provided by this specification for typically needed graphical information that is not interchanged, such as shape and line styles. This shown in Figure B.1 by the box labeled “DG” on the right, and by the box labeled “CS Mapping Specification” in the middle section. The arrow at the bottom of the middle section illustrates mappings being carried out according to the specification above it, producing a model of diagram visuals, or directly rendering the visuals on a display. Languages specifying this mapping reduce ambiguity and non uniformity in how their abstract syntax appears visually. The DG model is not expected to be specialized, enabling implementations to render instances of DG elements for all applications of the DD architecture. This specification provides normative CMOF and XSD artifacts for DG.

In the BPMN specification, the only realized part of the DD architecture so far is diagram interchange. Hence the only documentation provided by this annex is for the Diagram Interchange (DI) package, in addition to the relevant subset of Diagram Common (DC) package, which captures common data structure definitions. The documentation for the Diagram Graphics (DG) package is not provided here.

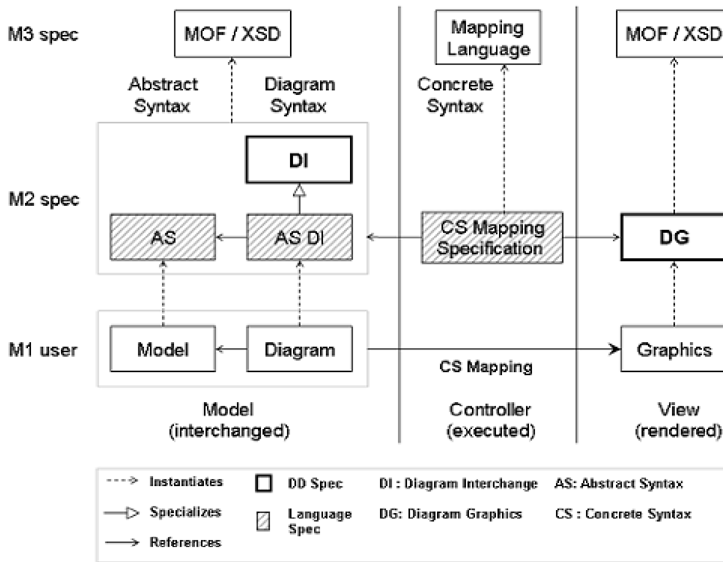


Figure B.1 – Diagram Definition Architecture

B.3 Diagram Common

The Diagram Common (DC) package contains abstractions shared by the Diagram Interchange and the Diagram Graphics packages.

B.3.1 Overview

The Diagram Common (DC) package contains a number of common primitive types as well as structured data types that are used in the definition of the Diagram Interchange (DG) package (see "Diagram Interchange" on page 487). The DC package itself does not depend on other packages. Some of the types defined in this package are defined based on similar ones in other related specifications including Cascading Style Sheets (CSS), Scalable Vector Graphics (SVG), and Office Document Format (ODF).

B.3.2 Abstract Syntax

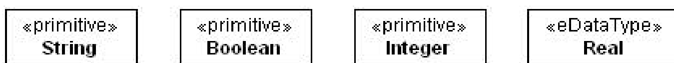


Figure B.2 – The Primitive Types

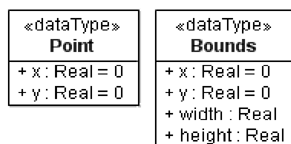


Figure B.3 – Diagram Definition Architecture

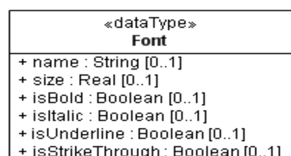


Figure B.4 – Diagram Definition Architecture

B.3.3 Classifier Descriptions

B.3.3.1 Boolean [PrimitiveType]

Boolean is a primitive data type having one of two values: true or false, intended to represent the truth value of logical expressions.

Description

Boolean is used as a type for typed elements that represent logical expressions. There are only two possible values for Boolean:

- true - The Boolean expression is satisfied.
- false - The Boolean expression is not satisfied.

Abstract Syntax

- Figure B.2 The primitive types

B.3.3.2 Bounds [PrimitiveType]

Bounds specifies an area in some (x, y) coordinate system that is enclosed by a bounded element's top-left point, its width, and its height.

Description

Bounds is used to specify the area of an element in some (x, y) coordinate system. The area is specified with a top-left point, representing the element's location (distance from the origin in logical units of length), in addition to the element's width and height (in logical units of length).

Abstract Syntax

- Figure B.3 (Layout Types)

Attributes

- + x : Real [1] = 0

  a real number that represents the x-coordinate of the rectangle.
- + y : Real[1] = 0

  a real number that represents the y-coordinate of the rectangle.
- + width : Real [1]

  a real number that represents the width of the rectangle.
- + height : Real [1]

  a real number that represents the height of the rectangle.

B.3.3.3 Font [PrimitiveType]

Font specifies the characteristics of a given font through a set of font properties.

Description

Font specifies a set of properties for a given font that is used when rendering text on a diagram

Abstract Syntax

- Figure B.4 The font type

Attributes

- + name : String[0..1]

  the name of the font (e.g., “Times New Roman,” “Arial,” and “Helvetica”).
- + size : Real [0..1]

  a non-negative real number representing the size of the font (expressed in the used unit of length).
- + isBold : Boolean [0..1]

  whether the font has a **bold** style.
- + isItalic : Boolean [0..1]

  whether the font has an *italic* style.
- + isUnderline : Boolean [0..1]

  whether the font has an underline style.
- + isStrikeThrough : Boolean [0..1]

  whether the font has a ~~strike-through~~ style.

B.3.3.4 Integer [PrimitiveType]

Integer is a primitive data type used to represent the mathematical concept of integer.

Description

Integer is used as a type for typed elements whose values are in the infinite set of integer numbers.

Abstract Syntax

- Figure B.2 The primitive types

B.3.3.5 Point [DataType]

A Point specifies an location in some (x, y) coordinate system.

Description

Point is used to specify a location in logical unit of length from the origin of some (x, y) coordinate system. The point (0, 0) is considered to be at the origin of that coordinate system.

Abstract Syntax

- Figure B.3 The layout types

Attributes

- + x : Real [1] = 0

  a real number that represents the x-coordinate of the point.

- + y : Real [1] = 0

  a real number that represents the y-coordinate of the point.

B.3.3.6 Real [PrimitiveType]

Real is a primitive data type used to represent the mathematical concept of real.

Description

Real is used as a type for typed elements whose values are in the infinite set of real numbers. Note that integer values are also considered real values and as such can be assigned to real-typed elements.

Abstract Syntax

- Figure B.2 The primitive types

B.3.3.7 String [PrimitiveType]

String is a primitive data type used to represent a sequence of characters in some suitable character set. Character sets may include both ASCII and Unicode characters.

Description

String is used as a type for typed elements in the metamodel that have text values. The allowed values for String depend on the semantics of the text in each context. A string value is a sequence of characters surrounded by double quotes (").

Abstract Syntax

- Figure B.2 The primitive types

B.4 Diagram Interchange

The Diagram Interchange (DI) package contains a model enabling interchange of graphical information that language users have control over, such as position of nodes and line routing points. Language specifications specialize elements of DI to define diagram interchange for a language.

B.4.1 Overview

The Diagram Interchange (DI) package contains a number of types used in the definition of diagram interchange models. The package imports the Diagram Common package (see "Diagram Common" on page 483), as shown in Figure B.5, that contains various relevant data types. The DI package contains mainly abstract types that are to be properly extended and refined by concrete types in domain-specific DI packages. In this sense, the DI package plays the role of a framework that is meant for extension rather than a component that is ready to be used out of the box. The benefit of this design is capture common assumptions in the DI package in order to facilitate the integration between various graphical domains that define their DI packages as extensions.

Diagrams are generally considered depictions of part or all of the elements in a domain-specific model. Therefore, one of the best practices adopted in the design of the DI package and that can be subsumed by the extending domain-specific DI packages is to minimize any redundancy with the depicted model when possible. For example, the text representing the name of a UML class is not defined as part of the UML class shape. This is primarily achieved by the fact that diagram elements reference their counterparts in the domain model as their context model elements instead of duplicating data from them. This design has the side effect of coupling the diagram models with their corresponding domain models, which is generally a common practice by tools. However, the DI package does not enforce this best practice and domain-specific DI packages can decide to have some level of duplication to decouple the models.

Another best practice adopted by the DI package is to avoid defining any data that is not changeable by the user but is rather derivable from the diagram's model context, like graphical rendering details. For example, the option to render a UML actor as a stick man or a as rectangle can be defined in a DI model as a boolean property to allow a user to choose between them. However, the definition of the actual line segments making up such shapes need not be interchanged in a DI model as it can be defined in the tool itself.

Other decisions that are left to the individual domain-specific DI packages include: whether to allow 1-n vs. m-n relationships between the domain elements and their referencing diagram elements, the formatting properties (styles) that affect the aesthetics of diagrams rather than their semantics that are allowed to be interchanged, and the degree of pragmatic redundancy that is allowed in the DI models to balance their footprint with the ease of their import/export.

B.4.2 Abstract Syntax

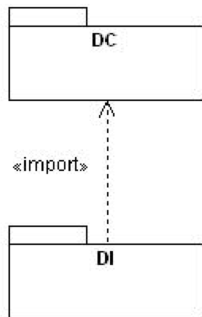


Figure B.5 – Dependencies of the DI package

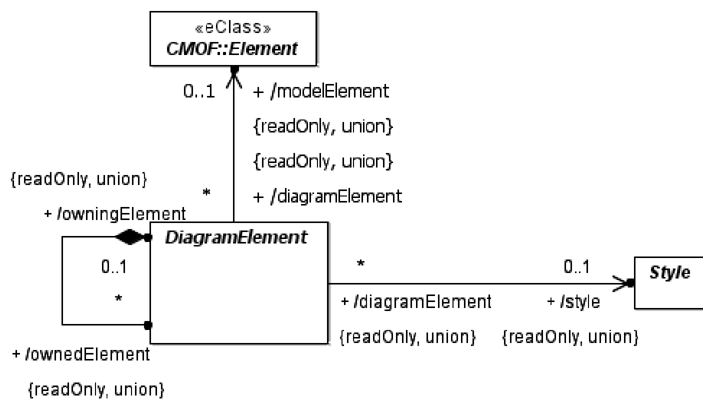


Figure B.6 – Diagram Element

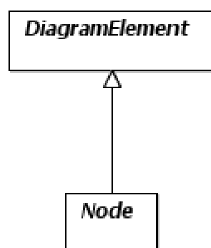


Figure B.7 – Node

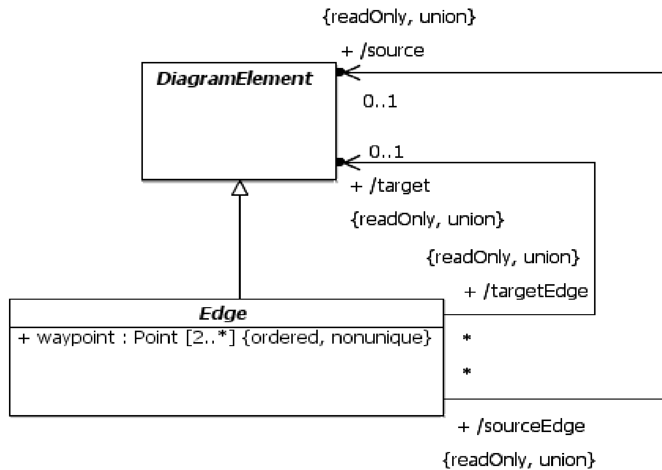


Figure B.8 – Edge

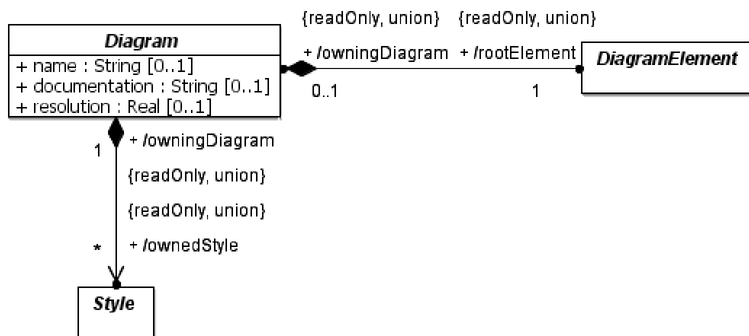


Figure B.9 – Diagram

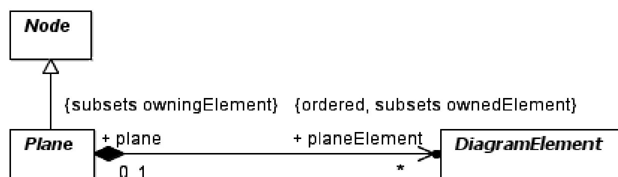


Figure B.10 – Plane

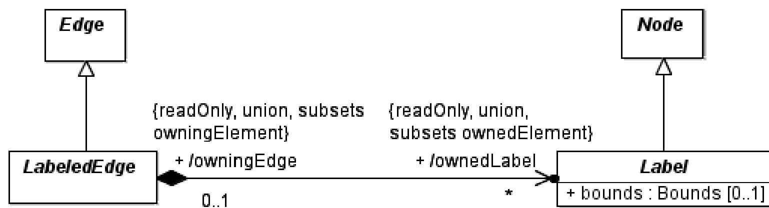


Figure B.11 – Labeled Edge

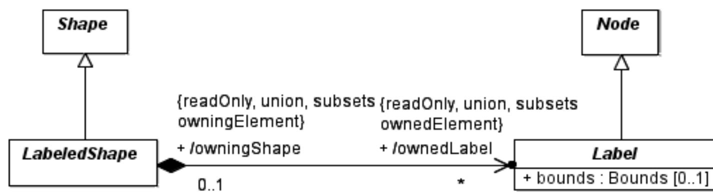


Figure B.12 – Labeled Shape

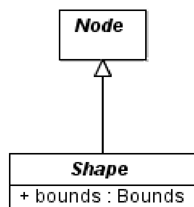


Figure B.13 – Shape

B.4.3 Classifier Descriptions

B.4.3.1 Diagram [Class]

Diagram is a container of a graph of diagram elements depicting all or part of a model.

Description

Diagram represents a depiction of all or part of a MOF model. A model can have one or more diagrams, each of which has a name and a description. A diagram contains the root of a graph of diagram elements that could reference various elements in a model. The root element is defined as a derived union, allowing domain-specific diagrams to specialize the root. All lengths specified by diagram elements are expressed in logical units of lengths. This unit of length would map to a unit of screen resolution (i.e., pixel) when rendering to the screen. To allow for predictable lengths when printing diagrams to paper, a diagram can also specify an intended printing resolution in Unit Per Inch (UPI). For example, a UPI of 300 means that a diagram element that is 300 unit wide would print as 1 inch wide on paper. A diagram can also own

a collection of styles that are referenced by its diagram elements. Styles contain unique combination of formatting properties used by different elements across the diagram. This allows for a large number of diagram elements to reference a small number of unique styles, which would dramatically reduce a diagram's footprint.

Abstract Syntax

- Figure B.9 Diagram

Attributes

- + name : String [0..1]

  the name of the diagram.

- + documentation : String [0..1]

  the documentation of the diagram.

- + resolution : Real [0..1]

the printing resolution of the diagram expressed in Unit Per Inch (UPI).

Associations

- ? + /rootElement : DiagramElement [1] {readOnly, union}

  the root of containment for all diagram elements contained in the diagram.

- ? + /ownedStyle : Style [*] {readOnly, union}

  the collection of styles owned by the diagram and referenced by its contained diagram elements.

B.4.3.2 DiagramElement [Class]

DiagramElement is the abstract supertype of all elements that can be nested in a diagram. It has two subtypes: Node and Edge.

Description

DiagramElement specifies an element that can be owned by a diagram and rendered to graphics. It is an abstract class that is further specialized by classes Node and Edge. A diagram element can either depict (reference) another context model element from an abstract syntax model (like UML or BPMN) or be purely notational (i.e., for enhancing the diagram understanding). In the case of depiction, data from both the diagram element and the model element are used for rendering. For example, the text of the name label of a UML class shape comes from the class, while the color of the label comes from the diagram element. A diagram element can reference a maximum of one model element, which can be any MOF-based element. The model element reference is a derived union and can be specialized in a domain-specific DI metamodel to be of a more concrete type.

Diagram elements can also own other diagram elements in a graph-like hierarchy. The collection of owned diagram elements is defined as a derived union. Domain-specific DI metamodels can specialize this collections to define what other diagram elements can be nested in a given diagram element.

Diagram elements can be specialized in a domain-specific DI metamodel to have domain-specific properties. Some of those properties augment the semantics of diagram elements and are therefore defied on the diagram elements. Other properties are considered formatting properties that influence the visual rendering of diagram elements but do not contribute to their semantics. Examples of such formatting properties include font, fill and stroke properties. Such properties tend to have similar values for diagram elements across the diagram and therefore to reduce the footprint of

diagrams, they are defined in Style elements that are owned by the diagram and referenced by individual diagram elements. For every unique combination of values for the style properties there would be a separate style element that is owned by the diagram. See “DiagramElement [Class]” on page 491 for more details.

There shall always be other properties that some tools wish to interchange that cannot be made normative. These can be interchanged using the extensibility mechanism that is native to the used interchange format (for example, an XSD schema following the XMI mapping would allow extraneous data to be placed on elements within <xmi:extension> tags, while a different XSD schema could allow this through xsd:any and xsd:anyAttribute elements placed in the definitions of extensible complex types).

Abstract Syntax

- Figure B.6 Diagram Element
- Figure B.7 Node
- Figure B.8 Edge
- Figure B.9 Diagram
- Figure B.10 Plane

Specializations

- Node
- Edge

Associations

- + /owningDiagram : Diagram [0..1] {readOnly, union}

  a reference to the diagram that directly owns this diagram element. The reference is only set for the root element in a diagram.

- + /owningElement : DiagramElement [0..1] {readOnly, union}

  a reference to the diagram element that directly owns this diagram element. The reference is set for all elements except the root element in a diagram.

- ? + /ownedElement : DiagramElement [*] {readOnly, union}

  a collection of diagram elements that are owned by this diagram element.

- + /modelElement : Element [0..1] {readOnly, union}

  a reference to a context model element, which can be any MOF-based element, for the diagram element.

- + /style : Style [0..1] {readOnly, union}

  a reference to an optional style containing formatting properties for the diagram element.

B.4.3.3 Edge [Class]

Edge specifies a given edge in a graph of diagram elements. It represents a polyline connection between two graph elements: a source and a target.

Description

Edge represents a given connection between two elements in a diagram, a source element and a target element. An edge often references a relationship element (like a UML generalization or a BPMN message flow) as a context model element. It can also be purely notational, i.e., does not reference any model element. When referencing a relationship model element, the edge's source and target reference the relationship's source and target respectively as their model elements. If the edge's source and target can be derived unambiguously from other info (like the edge's model element or the edge's class type), they are not explicitly set on the edge to avoid redundancy, otherwise they need to be set. The source and target are defined as derived unions to allow domain-specific DI metamodels to specialize them appropriately.

An edge is often depicted as a line with 2 or more points (i.e., one or more connected line segments) in the coordinate system, called waypoints. The first point typically intersects with the edge's source, while the last point typically intersects with the edge's target. Any points in between establish a route for the line to traverse in the diagram.

Abstract Syntax

- Figure B.8 Edge
- Figure B.11 Labeled Edge

Generalizations

- DiagramElement

Specializations

- LabeledEdge

Attributes

- + waypoint : Point [2..*] {ordered, nonunique}

  a list of two or more points relative to the origin of the coordinate system (e.g., the origin of a containing plane) that specifies the connected line segments of the edge.

Associations

- + /source : DiagramElement [0..1] {readOnly, union}

  the edge's source diagram element, i.e., where the edge starts from. It is optional and needs to be set only if it cannot be unambiguously derived.

- + /target : DiagramElement [0..1] {readOnly, union}

  the edge's target diagram element, i.e., where the edge ends at. It is optional and needs to be set only if it cannot be unambiguously derived.

B.4.3.4 Label [Class]

Label represents a node that is owned by another main diagram element in a plane and that depicts some (usually textual) aspect of that element within its own separate bounds.

Description

Label represents an owned node of another diagram element, typically a LabeledShape or a LabeledEdge. A label typically depicts some (usually textual) aspect of its owning element that needs to be laid out separately using the label's own bounds. The bounds are optional and if not specified, the label will be positioned in its default position.

A label's model element is typically not specified as it can be derived from its owning element. However, if the model element cannot be unambiguously derived, then a label could be given ts own separate model element to disambiguate it.

Abstract Syntax

- Figure B.11 (Labeled Edge)
- Figure B.12 Labeled Shape

Generalizations

- Node

Attributes

- + bounds : Bounds [1]

  the bounds (x, y, width and height) of the label relative to the origin of a containing plane.

B.4.3.5 LabeledEdge [Class]

LabeledEdge represents an edge that owns a collection of labels.

Description

LabeledEdge is an edge that owns a collection of labels (see "LabeledEdge [Class]" on page 494) that depict some aspects of it. An example is a UML association that has a number of labels (e.g., a name label, two role name labels and two multiplicity labels) positioned beside it. The existence of a label in this collection specifies that it is visible. The separate optional bounds of the label indicate where it should be positioned and if not specified the label can be positioned in its default position.

Abstract Syntax

- Figure B.11 Labeled Edge

Generalizations

- Edge

Associations

- ? + /ownedLabel : Label [*] {readOnly, union, subsets ownedNode}

  the collection of labels owned by this edge.

B.4.3.6 LabeledShape [Class]

LabeledShape represents a shape that owns a collection of labels.

Description

LabeledShape is a shape that owns a collection of labels (see "LabeledShape [Class]" on page 494) that depict some aspects of it. An example is a UML port shape that is rendered as a filled box and has a name label positioned beside it. The existence of a label in this collection specifies that it is visible. The separate optional bounds of the label indicate where it should be positioned and if not specified the label can be positioned in its default position.

Abstract Syntax

- Figure B.12 Labeled Shape

Generalizations

- Shape

Associations

- ? + /ownedLabel : Label [*] {readOnly, union, subsets ownedNode}
  the collection of labels owned by this shape.

B.4.3.7 Node [Class]

Node specifies a given node in a graph of diagram elements.

Description

Node represents a given node (or vertex) in a diagram, which is a graph of diagram elements. A node often references a non-relationship element (like a UML class or a BPMN activity) as a model element. It can also be purely notational, i.e., does not reference any model element.

The abstract node class does not have any particular layout characteristics. However, it may gets specialized in a domain-specific DI metamodel to define nodes that have certain layout characteristics. Examples include planes with infinite bounds, shapes with limited bounds, tree items and graph vertices...etc.

Abstract Syntax

- Figure B.7 Node
- Figure B.10 Plane
- Figure B.11 Labeled Edge
- Figure B.12 Labeled Shape
- Figure B.13 Shape

Generalizations

- DiagramElement

Specializations

- Label
- Shape
- Plane

B.4.3.8 Plane [Class]

Plane is a node with an infinite bounds in the x-y coordinate system that owns a collection of shapes and edges that are laid out relative to its origin point.

Description

Plane has an origin point (0, 0) and an infinite size along the x and y axes. The coordinate system of the plane increases along the x-axis from left to right and along the y-axis from top to bottom. All the nested shapes and edges are laid out relative to their plane's origin.

A plane is often chosen as a root element for a two dimensional diagram that depicts an inter-connected graph of shapes an edges. A plane may have its own reference to a model element, in which case the whole plane is considered a depiction of that element. Alternatively, a plane without a reference to a model element is simply a layout container for its shapes and edges.

The collection of plane elements (shapes and edges) in a plane is ordered with the order specifying the z-order of these plane elements relative to each other. The higher the z-order, the more to the front (on top) the plane element is.

Abstract Syntax

- Figure B.10 Plane

Generalizations

- Node

Associations

- ? + planeElement : DiagramElement [*] {subsets ownedNode}

  the ordered collection of diagram elements owned by this plane with the order defining the z-order of the diagram element.

B.4.3.9 Shape [Class]

Shape represents a node that has bounds that is relevant to the origin of a containing plane.

Description

Shape represents a node that is directly or indirectly owned by a plane (See "Shape [Class]" on page 496.) and that is laid out according to a given bounds that is relevant to the origin of the plane. A shape does not have any particular graphical rendering, i.e., the rendering is domain-specific.

A shape can be purely notational (i.e., does not reference any model element), like a block arrow pointing to a UML class shape with some textual message or an overlay rectangle with some transparent fill enclosing a bunch of shapes on the diagram to make them stand out. However, a shape often represents a depiction of a non-relational element from a business model (like UML class or BPMN activity) and hence references such an element as its model element.

Abstract Syntax

- Figure B.13 Shape
- Figure B.12 Labeled Shape

Generalizations

- Node

Specializations

- LabeledShape

Attributes

- + bounds : Bounds [1]

  the bounds (x, y, width and height) of the shape relative to the origin of a containing plane.

B.4.3.10 Style [Class]

A style is a container for a collection of properties that affect the formatting of a set of diagram elements rather than their structure or semantics.

Description

A style represents a bag of properties that affect the appearance of a group of diagram elements. A style property (like font, fill, or stroke) is distinguishable from a property on a diagram element in that it is meant for the aesthetics of the element rather than being part of its intrinsic syntax.

A style tends to have only a few unique value combinations for its properties across the diagram. Such combinations are represented by different style instances owned by the diagram and referenced by the diagram elements. This allows for conserving the footprint of diagrams (over making style instances owned by diagram elements).

Style is defined as an abstract class without prescribing any style properties to leave it up to domain-specific DI metamodels to define concrete style classes that are applicable to their diagram element types.

Abstract Syntax

- Figure B.6 Diagram Element
- Figure B.9 Diagram

Annex C
Glossary

(informative)

A

Activity
Work that a company or organization performs using business processes. An activity can be atomic or non-atomic (compound). The types of activities that are a part of a Process Model are: Process, Sub-Process, and Task.

Abstract Process
A Process that represents the interactions between a private business process and another process or participant.

Artifact
A graphical object that provides supporting information about the Process or elements within the Process. However, it does not directly affect the flow of the Process.

Association
A connecting object that is used to link information and Artifacts with Flow Objects. An association is represented as a dotted graphical line with an arrowhead to represent the direction of flow.

Atomic Activity
An activity not broken down to a finer level of Process Model detail. It is a leaf in the tree-structure hierarchy of Process activities. Graphically it will appear as a Task in BPMN.

B

Business Analyst
A specialist who analyzes business needs and problems, consults with users and stakeholders to identify opportunities for improving business return through information technology, and defines, manages, and monitors the requirements into business processes.

Business Process
A defined set of business activities that represent the steps required to achieve a business objective. It includes the flow and use of information and resources.

Business Process Management
The services and tools that support process management (for example, process analysis, definition, processing, monitoring and administration), including support for human and application-level interaction. BPM tools can eliminate manual processes and automate the routing of requests between departments and applications.

BPM System
The technology that enables BPM.

C

Choreography
An ordered sequence of B2B message exchanges between two or more Participants. In a Choreography there is no central controller, responsible entity, or observer of the Process.

Collaboration
Collaboration is the act of sending messages between any two Participants in a BPMN model. The two Participants represent two separate BPML processes.

Collapsed Sub-Process
A Sub-Process that hides its flow details. The Collapsed Sub-Process object uses a marker to distinguish it as a Sub-Process, rather than a Task. The marker is a small square with a plus sign (+) inside.

Compensation Flow
Flow that defines the set of activities that are performed while the transaction is being rolled back to compensate for activities that were performed during the Normal Flow of the Process. A Compensation Flow can also be called from a Compensate End or Intermediate Event.

Compound Activity
An activity that has detail that is defined as a flow of other activities. It is a branch (or trunk) in the tree-structure hierarchy of Process activities. Graphically, it will appear as a Process or Sub-Process in BPMN.

Controlled Flow
Flow that proceeds from one Flow Object to another, via a Sequence Flow link, but is subject to either conditions or dependencies from other flow as defined by a Gateway. Typically, this is seen as a Sequence flow between two activities, with a conditional indicator (mini-diamond) or a Sequence Flow connected to a Gateway.

D

Decision
A gateway within a business process where the Sequence Flow can take one of several alternative paths. Also known as "Or-Split."

E

End Event
An Event that indicates where a path in the process will end. In terms of Sequence Flows, the End Event ends the flow of the Process, and thus, will not have any outgoing Sequence Flows. An End Event can have a specific Result that will appear as a marker within the center of the End Event shape. End Event Results are Message, Error, Compensation, Signal, Link, and Multiple. The End Event shares the same basic shape of the Start Event and Intermediate Event, a circle, but is drawn with a thick single line.

Event Context
An Event Context is the set of activities that can be interrupted by an exception (Intermediate Event). This can be one activity or a group of activities in an expanded Sub-Process.

Exception
An event that occurs during the performance of the Process that causes a diversion from the Normal Flow of the Process. Exceptions can be generated by Intermediate Events, such as time, error, or message.

Exception Flow
A Sequence Flow path that originates from an Intermediate Event attached to the boundary of an activity. The Process does not traverse this path unless the Activity is interrupted by the triggering of a boundary Intermediate Event (an Exception - see above).

Expanded Sub-Process
A Sub-Process that exposes its flow detail within the context of its Parent Process. An Expanded Sub-Process is displayed as a rounded rectangle that is enlarged to display the Flow Objects within.

F

Flow
A directional connector between elements in a Process, Collaboration, or Choreography. A Sequence Flows represents the sequence of Flow Objects in a Process or Choreography. A Message Flow represents the transmission of a Message between Collaboration Participants.The term Flow is often used to represent the overall progression of how a Process or Process segment would be performed.

Flow Object
A graphical object that can be connected to or from a Sequence Flow. In a Process, Flow Objects are Events, Activities, and Gateways. In a Choreography, Flow Objects are Events, Choreography Activities, and Gateways.

Fork A point in the Process where one Sequence Flow path is split into two or more paths that are run in parallel within the Process, allowing multiple activities to run simultaneously rather than sequentially. BPMN uses multiple outgoing Sequence Flows from Activities or Events or a Parallel Gateway to perform a Fork. Also known as "AND-Split."

I

Intermediate Event An event that occurs after a Process has been started. An Intermediate Event affects the flow of the process by showing where messages and delays are expected, distributing the Normal Flow through exception handling, or showing the extra flow required for compensation. However, an Intermediate Event does not start or directly terminate a process. An Intermediate Event is displayed as a circle, drawn with a thin double line.

J

Join A point in the Process where two or more parallel Sequence Flow paths are combined into one Sequence Flow path. BPMN uses a Parallel Gateway to perform a Join. Also known as "AND-Join."

L

Lane A partition that is used to organize and categorize activities within a Pool. A Lane extends the entire length of the Pool either vertically or horizontally. Lanes are often used for such things as internal roles (e.g., Manager, Associate), systems (e.g., an enterprise application), or an internal department (e.g., shipping, finance).

M

Merge A point in the Process where two or more alternative Sequence Flow paths are combined into one Sequence Flow path. No synchronization is required because no parallel activity runs at the join point. BPMN uses multiple incoming Sequence Flows for an Activity or an Exclusive Gateway to perform a Merge. Also know as "OR-Join."

Message An Object that depicts the contents of a communication between two Participants. A message is transmitted through a Message Flow and has an identity that can be used for alternative branching of a Process through the Event-Based Exclusive Gateway.

Message Flow A Connecting Object that shows the flow of messages between two Participants. A Message Flow is represented by a dashed lined.

N

Normal Flow A flow that originates from a Start Event and continues through activities on alternative and parallel paths until reaching an End Event.

P

Parent Process A Process that holds a Sub-Process within its boundaries.

Participant A business entity (e.g., a company, company division, or a customer) or a business role (e.g., a buyer or a seller) that controls or is responsible for a business process. If Pools are used, then a Participant would be associated with one Pool. In a Collaboration, Participants are informally known as "Pools."

Pool A Pool represents a Participant in a Collaboration. Graphically, a Pool is a container for partitioning a Process from other Pools/Participants. A Pool is not required to contain a Process, i.e., it can be a "black box."

Private Business Process A process that is internal to a specific organization and is the type of process that has been generally called a workflow or BPM process.

Process A sequence or flow of Activities in an organization with the objective of carrying out work. In BPMN, a Process is depicted as a graph of Flow Elements, which are a set of Activities, Events, Gateways, and Sequence Flow that adhere to a finite execution semantics.

R

Result The consequence of reaching an End Event. Types of Results include Message, Error, Compensation, Signal, Link, and Multiple.

S

Sequence Flow A connecting object that shows the order in which activities are performed in a Process and is represented with a solid graphical line. Each Flow has only one source and only one target. A Sequence Flow can cross the boundaries between Lanes of a Pool but cannot cross the boundaries of a Pool.

Start Event An Event that indicates where a particular Process starts. The Start Event starts the flow of the Process and does not have any incoming Sequence Flow, but can have a Trigger. The Start Event is displayed as a circle, drawn with a single thin line.

Sub-Process A Process that is included within another Process. The Sub-Process can be in a collapsed view that hides its details. A Sub-Process can be in an expanded view that shows its details within the view of the Process that it is contained in. A Sub-Process shares the same shape as the Task, which is a rectangle that has rounded corners.

Swimlane A Swimlane is a graphical container for partitioning a set of activities from other activities. BPMN has two different types of Swimlanes. See “Pool” and “Lane.”

T

Task An atomic activity that is included within a Process. A Task is used when the work in the Process is not broken down to a finer level of Process Model detail. Generally, an end-user, an application, or both will perform the Task. A Task object shares the same shape as the Sub-Process, which is a rectangle that has rounded corners.

Token A theoretical concept that is used as an aid to define the behavior of a Process that is being performed. The behavior of Process elements can be defined by describing how they interact with a token as it “traverses” the structure of the Process. For example, a token will pass through an Exclusive Gateway, but continue down only one of the Gateway's outgoing Sequence Flow.

Transaction A Sub-Process that represents a set of coordinated activities carried out by independent, loosely-coupled systems in accordance with a contractually defined business relationship. This coordination leads to an agreed, consistent, and verifiable outcome across all participants.

Trigger A mechanism that detects an occurrence and can cause additional processing in response, such as the start of a business Process. Triggers are associated with Start Events and Intermediate Events and can be of the type: Message, Timer, Conditional, Signal, Link, and Multiple.

U

Uncontrolled Flow Flow that proceeds without dependencies or conditional expressions. Typically, an Uncontrolled Flow is a Sequence Flow between two Activities that do not have a conditional indicator (mini-diamond) or an intervening Gateway.