Business Process Model and Notation (BPMN)

Version 2.0
with change bars

OMG Document Number: formal/2011-01-04
Standard document URL: http://www.omg.org/spec/BPMN/2.0
Associated Schema Files:

  XMI: BPMN20.cmofof
       BPMNDI.cmofof
       DC.cmofof
       DI.cmofof
  XSD: BPMN20.xsd
       BPMNDI.xsd
       DC.xsd
       DI.xsd
       Semantic.xsd
  XSLT: BPMN20-FromXML.xslt
        BPMN20-ToXML.xslt

  Infrastructure.cmofof
  Semantic.cmofof
USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.
GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 140 Kendrick Street, Needham, MA 02494, U.S.A.

TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™, Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™, IMM™, MOF™, OMG Interface Definition Language (IDL)™, and OMG Systems Modeling Language (OMG SysML)™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.
COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.
OMG’s Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).
## Table of Contents

Preface .................................................................................................................. xxiii  
1 Scope .................................................................................................................. 1  
2 Conformance ...................................................................................................... 1  
  2.1 Process Modeling Conformance ..................................................................... 2  
    2.1.1 BPMN Process Types ............................................................................. 2  
    2.1.2 BPMN Process Elements ..................................................................... 2  
    2.1.3 Visual Appearance ................................................................................ 8  
    2.1.4 Structural Conformance ...................................................................... 8  
    2.1.5 Process Semantics ............................................................................... 8  
    2.1.6 Attributes and Model Associations .................................................... 9  
    2.1.7 Extended and Optional Elements ....................................................... 9  
    2.1.8 Visual Interchange .............................................................................. 9  
  2.2 Process Execution Conformance .................................................................. 9  
    2.2.1 Execution Semantics .......................................................................... 10  
    2.2.2 Import of Process Diagrams ............................................................... 10  
  2.3 BPEL Process Execution Conformance ................................................... 10  
  2.4 Choreography Modeling Conformance ...................................................... 10  
    2.4.1 BPMN Choreography Types ............................................................... 10  
    2.4.2 BPMN Choreography Elements ......................................................... 10  
    2.4.3 Visual Appearance ............................................................................. 11  
    2.4.4 Choreography Semantics ................................................................... 11  
    2.4.5 Visual Interchange ............................................................................. 11  
  2.5 Summary of BPMN Conformance Types .................................................. 12  
3 Normative References ....................................................................................... 12  
  3.1 Normative .................................................................................................... 12  
  3.2 Non-Normative ........................................................................................... 13  
4 Terms and Definitions ....................................................................................... 16  
5 Symbols .............................................................................................................. 16  
6 Additional Information ....................................................................................... 16  
  6.1 Conventions ................................................................................................ 16  
    6.1.1 Typographical and Linguistic Conventions and Style ......................... 16  
    6.1.2 Abbreviations ..................................................................................... 17  
  6.2 Structure of this Document ........................................................................ 17  
  6.3 Acknowledgements ..................................................................................... 17  
7 Overview ............................................................................................................ 21
List of Figures

Figure 7.1 - Example of a private Business Process 23
Figure 7.2 - Example of a public Process 24
Figure 7.3 - An example of a Collaborative Process 25
Figure 7.4 - An example of a Choreography 25
Figure 7.5 - An example of a Conversation diagram 26
Figure 7.6 - An example of a Collaboration diagram with black-box Pools 45
Figure 7.7 - An example of a stand-alone Choreography diagram 46
Figure 7.8 - An example of a stand-alone Process (Orchestration) diagram 47
Figure 8.1 - A representation of the BPMN Core and Layer Structure 49
Figure 8.2 - Class diagram showing the core packages 50
Figure 8.3 - Class diagram showing the organization of the core BPMN elements 51
Figure 8.4 - Definitions class diagram 52
Figure 8.5 - Classes in the Foundation package 55
Figure 8.6 - Extension class diagram 57
Figure 8.7 - External Relationship Metamodel 62
Figure 8.8 - Artifacts Metamodel 66
Figure 8.9 - An Association 67
Figure 8.10 - The Association Class Diagram 67
Figure 8.11 - A Directional Association 68
Figure 8.12 - An Association of Text Annotation 68
Figure 8.13 - A Group Artifact 69
Figure 8.14 - A Group around Activities in different Pools 69
Figure 8.15 - The Group class diagram 70
Figure 8.16 - A Text Annotation 71
Figure 8.17 - The Correlation Class Diagram 76
Figure 8.18 - Error class diagram 81
Figure 8.19 - Escalation class diagram 82
Figure 8.20 - Event class diagram 84
Figure 8.21 - Expression class diagram 85
Figure 8.22 - FlowElement class diagram 87
Figure 8.23 - FlowElementContainers class diagram 89
Figure 8.24 - Gateway class diagram 90
Figure 8.25 - ItemDefinition class diagram 92
Figure 8.26 - A Message 93
Figure 8.27 - A non-initiating Message 93
Figure 8.28 - Messages Association overlapping Message Flows 94
Figure 8.29 - Messages shown Associated with a Choreography Task 94
Figure 8.30 - The Message class diagram 95
Figure 8.31 - Resource class diagram 96
Figure 8.32 - A Sequence Flow 97
Figure 8.33 - A Conditional Sequence Flow 97
Figure 8.34 - A Default Sequence Flow 98
Figure 8.35 - SequenceFlow class diagram 98
Figure 8.36 - The Service class diagram 104
Figure 9.1 - Classes in the Collaboration package 109
Figure 9.2 - A Pool 112
Figure 9.3 - Message Flows connecting to the boundaries of two Pools 113
Figure 9.4 - Message Flows connecting to Flow Objects within two Pools 113
Figure 9.5 - Main (Internal) Pool without boundaries 114
Figure 9.6 - Pools with a Multi-Instance Participant Markers 114
Figure 9.7 - The Participant Class Diagram 115
Figure 9.8 - A Pool with a Multiple Participant 117
Figure 9.9 - The Participant Multiplicity class diagram 117
Figure 9.10 - ParticipantAssociation class diagram 119
Figure 9.11 - A Message Flow 120
Figure 9.12 - A Message Flow with an Attached Message 121
Figure 9.13 - A Message Flow passing through a Choreography Task 121
Figure 9.14 - The Message Flow Class Diagram 122
Figure 9.15 - MessageFlowAssociation class diagram 124
Figure 9.16 - A Conversation diagram 125
Figure 9.17 - A Conversation diagram where the Conversation is expanded into Message Flows 125
Figure 9.18 - Conversation diagram depicting several conversations between Participants in a related domain 126
Figure 9.19 - An example of a Sub-Conversation 127
Figure 9.20 - An example of a Sub-Conversation expanded to a Conversation and Message Flow 127
Figure 9.21 - An example of a Sub-Conversation that is fully expanded 128
Figure 9.22 - Metamodel of ConversationNode Related Elements 129
Figure 9.23 - A Communication element 130
Figure 9.24 - A compound Conversation element 131
Figure 9.25 - A Call Conversation calling a GlobalConversation 131
Figure 9.26 - A Call Conversation calling a Collaboration 131
Figure 9.27 - A Conversation Link element 132
Figure 9.28 - Conversation links to Activities and Events 133
Figure 9.29 - Metamodel of Conversation Links related elements 134
Figure 9.30 - Call Conversation Links 135
Figure 9.31 - The ConversationAssociation class diagram 136
Figure 9.32 - An example of a Choreography within a Collaboration 138
Figure 9.33 - Choreography within Collaboration class diagram 139
Figure 10.1 - An Example of a Process 145
Figure 10.2 - Process class diagram 146
Figure 10.3 - Process Details class diagram 147
Figure 10.4 - Example of a private Business Process 150
Figure 10.5 - Example of a public Process 150
Figure 10.6 - Activity class diagram 151
Figure 10.7 - The class diagram for assigning Resources 154
Figure 10.8 - A Task object 156
Figure 10.9 - Task markers 157
Figure 10.10 - The Task class diagram 157
Figure 10.11 - A Service Task Object 158
Figure 10.12 - The Service Task class diagram 159
Figure 10.13 - A Send Task Object 160
Figure 10.14 - The Send Task and Receive Task class diagram 160
Figure 10.15 - A Receive Task Object 161
Figure 10.16 - A Receive Task Object that instantiates a Process 162
Figure 10.17 - A User Task Object 163
Figure 10.18 - A Manual Task Object 163
Figure 10.19 - A Business Rule Task Object 164
Figure 10.20 - A Script Task Object 164
Figure 10.21 - Manual Task class diagram 166
Figure 10.22 - User Task class diagram 166
Figure 10.23 - HumanPerformer class diagram 168
Figure 10.24 - Procurement Process Example 170
Figure 10.25 - A Sub-Process object (collapsed) 174
Figure 10.26 - A Sub-Process object (expanded) 174
Figure 10.27 - Expanded Sub-Process used as a “Parallel Box” 174
Figure 10.28 - Collapsed Sub-Process Markers 175
Figure 10.29 - The Sub-Process class diagram 176
Figure 10.30 - An Event Sub-Process object (Collapsed) 177
Figure 10.31 - An Event Sub-Process object (expanded) 177
Figure 10.32 - An example that includes Event Sub-Processes 178
Figure 10.33 - A Transaction Sub-Process 179
Figure 10.34 - A Collapsed Transaction Sub-Process 179
Figure 10.35 - A collapsed Ad-Hoc Sub-Process 181
Figure 10.36 - An expanded Ad-Hoc Sub-Process 181
Figure 10.37 - An Ad-Hoc Sub-Process for writing a book chapter 182
Figure 10.38 - An Ad-Hoc Sub-Process with data and sequence dependencies 183
Figure 10.39 - A Call Activity object calling a Global Task 184
Figure 10.40 - A Call Activity object calling a Process (Collapsed) 184
Figure 10.41 - A Call Activity object calling a Process (Expanded) 184
Figure 10.42 - The Call Activity class diagram 185
Figure 10.43 - CallableElement class diagram 186
Figure 10.44 - Global Tasks class diagram 188
Figure 10.45 - LoopCharacteristics class diagram 189
Figure 10.46 - A Task object with a Standard Loop Marker 190
Figure 10.47 - A Sub-Process object with a Standard Loop Marker 190
Figure 10.48 - Activity Multi-Instance marker for parallel instances 191
Figure 10.49 - Activity Multi-Instance marker for sequential instances 191
Figure 10.50 - ItemAware class diagram 204
Figure 10.51 - DataObject class diagram 205
Figure 10.52 - A DataObject 207
Figure 10.53 - A DataObject that is a collection 207
Figure 10.54 - A Data Store 208
Figure 10.55 - DataStore class diagram 209
Figure 10.56 - Property class diagram 210
Figure 10.57 - InputOutputSpecification class diagram 212
Figure 10.58 - A DataInput 214
Figure 10.59 - Data Input class diagram 214
Figure 10.60 - A Data Output 216
Figure 10.61 - Data Output class diagram 216
Figure 11.27 - The Call Choreography class diagram 334
Figure 11.28 - A valid sequence of Choreography Activities 336
Figure 11.29 - The corresponding Collaboration for a valid Choreography sequence 337
Figure 11.30 - A valid sequence of Choreography Activities with a two-way Activity 337
Figure 11.31 - The corresponding Collaboration for a valid Choreography sequence with a two-way Activity 338
Figure 11.32 - An invalid sequence of Choreography Activities 338
Figure 11.33 - The corresponding Collaboration for an invalid Choreography sequence 339
Figure 11.34 - An example of the Exclusive Gateway 346
Figure 11.35 - The relationship of Choreography Activity Participants across the sides of the Exclusive Gateway shown through a Collaboration 347
Figure 11.36 - Different Receiving Choreography Activity Participants on the output sides of the Exclusive Gateway 348
Figure 11.37 - The corresponding Collaboration view of the above Choreography Exclusive Gateway configuration 349
Figure 11.38 - An example of an Event Gateway 350
Figure 11.39 - The corresponding Collaboration view of the above Choreography Event Gateway configuration 351
Figure 11.40 - An example of a Choreography Inclusive Gateway configuration 353
Figure 11.41 - The corresponding Collaboration view of the above Choreography Inclusive Gateway configuration 354
Figure 11.42 - An example of a Choreography Inclusive Gateway configuration 355
Figure 11.43 - The corresponding Collaboration view of the above Choreography Inclusive Gateway configuration 356
Figure 11.44 - Another example of a Choreography Inclusive Gateway configuration 357
Figure 11.45 - The corresponding Collaboration view of the above Choreography Inclusive Gateway configuration 358
Figure 11.46 - The relationship of Choreography Activity Participants across the sides of the Parallel Gateway 359
Figure 11.47 - The corresponding Collaboration view of the above Choreography Parallel Gateway configuration 360
Figure 11.48 - An example of a Choreography Complex Gateway configuration 361
Figure 11.49 - The corresponding Collaboration view of the above Choreography Complex Gateway configuration 362
Figure 11.50 - An example of a Choreography Process combined with Black Box Pools 363
Figure 11.51 - An example of a Choreography Process combined with Pools that contain Processes 364
Figure 12.1 - BPMN Diagram 369
Figure 12.2 - BPMN Plane 369
Figure 12.3 - BPMN Shape 369
Figure 12.4 - BPMN Edge 370
Figure 12.5 - BPMN Label 370
Figure 12.6 - Depicting a Label for a DataObjectReference with its state 381
Figure 12.7 - Combined Compensation and Loop Characteristic Marker Example 384
Figure 12.8 - Expanded Sub-Process Example 413
Figure 12.9 - Start and End Events on the Border Example 415
Figure 12.10 - Collapsed Sub-Process 416
Figure 12.11 - Contents of Collapsed Sub-Process 416
Figure 12.12 - Nested Lanes Example 418
Figure 12.13 - Vertical Collaboration Example 419
Figure 12.14 - Conversation Example 420
Figure 12.15 - Choreography Example 422
Figure 13.1 - Behavior of multiple outgoing Sequence Flows of an Activity 427
Figure 13.2 - The Lifecycle of a BPMN Activity 428
Figure 13.3 - Merging and Branching Sequence Flows for a Parallel Gateway 434
Figure 13.4 - Merging and Branching Sequence Flows for an Exclusive Gateway 435
Figure 13.5 - Merging and Branching Sequence Flows for an Inclusive Gateway 435
Figure 13.6 - Merging and branching Sequence Flows for an Event-Based Gateway 437
Figure 13.7 - Merging and branching Sequence Flows for a Complex Gateway 437
Figure 14.1 - A BPMN orchestration process and its block hierarchy 446
Figure 14.2 - An example of distributed token recombination 469
Figure 14.3 - An example of a loop from a decision with more than two alternative paths 470
Figure 14.4 - An example of interleaved loops 471
Figure 14.5 - An example of the WSBPEL pattern for substituting for the derived Process 472
Figure 14.6 - An example of a WSBPEL pattern for the derived Process 472
Figure 14.7 - An example – An infinite loop 473
Figure 14.8 - An example - Activity that spans two paths of a WSBPEL structured element 474
Figure B.1 - Diagram Definition Architecture 483
Figure B.2 - The Primitive Types 483
Figure B.3 - Diagram Definition Architecture 484
Figure B.4 - Diagram Definition Architecture 484
Figure B.5 - Dependencies of the DI package 488
Figure B.6 - Diagram Element 488
Figure B.7 - Node 488
Figure B.8 - Edge 489
Figure B.9 - Diagram 489
Figure B.10 - Plane 489
Figure B.11 - Labeled Edge 490
Figure B.12 - Labeled Shape 490
Figure B.13 - Shape 490
List of Tables

Table 2.1 – Descriptive Conformance Sub-Class Elements and Attributes 3
Table 2.2 – Analytic Conformance Sub-Class Elements and Attributes 4
Table 2.3 – Common Executable Conformance Sub-Class Elements and Attributes 6
Table 2.4 – Common Executable Conformance Sub-Class Supporting Classes 7
Table 2.5 – Types of BPMN Conformance 12
Table 7.1 – Basic Modeling Elements 29
Table 7.2 – BPMN Extended Modeling Elements 31
Table 7.3 – Sequence Flow Connection Rules 42
Table 7.4 – Message Flow Connection Rules 44
Table 8.1 – Definitions attributes and model associations 53
Table 8.2 – Import attributes 54
Table 8.3 – Definitions XML schema 54
Table 8.4 – Import XML schema 55
Table 8.5 – BaseElement attributes and model associations 56
Table 8.6 – Documentation attributes 56
Table 8.7 – Extension attributes and model associations 58
Table 8.8 – ExtensionDefinition attributes and model associations 59
Table 8.9 – ExtensionAttributeDefinition attributes 59
Table 8.10 – ExtensionAttributeValue model associations 59
Table 8.11 – Extension XML schema 60
Table 8.12 – Example Core XML schema 60
Table 8.13 – Example Extension XML schema 61
Table 8.14 – Sample XML instance 61
Table 8.15 – Relationship attributes 63
Table 8.16 – Reengineer XML schema 63
Table 8.17 – BaseElement XML schema 64
Table 8.18 – RootElement XML schema 65
Table 8.19 – Relationship XML schema 65
Table 8.20 – Association attributes and model associations 68
Table 8.21 – Group model associations 70
Table 8.22 – Category model associations 71
Table 8.23 – CategoryValue attributes and model associations 71
Table 8.24 – Text Annotation attributes 72
Table 8.25 – Artifact XML schema 72
Table 8.26 – Association XML schema 72
Table 8.27 – Category XML schema 72
Table 8.28 – CategoryValue XML schema 73
Table 8.29 – Group XML schema 73
Table 8.30 – Text Annotation XML schema 73
Table 8.31 – CorrelationKey model associations 77
Table 8.32 – CorrelationProperty model associations 77
Table 8.33 – CorrelationPropertyRetrievalExpression model associations 78
Table 8.34 – CorrelationSubscription model associations 78
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.35</td>
<td>CorrelationPropertyBinding model associations</td>
<td>79</td>
</tr>
<tr>
<td>8.36</td>
<td>Correlation Key XML schema</td>
<td>79</td>
</tr>
<tr>
<td>8.37</td>
<td>Correlation Property XML schema</td>
<td>79</td>
</tr>
<tr>
<td>8.38</td>
<td>Correlation Property Binding XML schema</td>
<td>80</td>
</tr>
<tr>
<td>8.39</td>
<td>Correlation Property Retrieval Expression XML schema</td>
<td>80</td>
</tr>
<tr>
<td>8.40</td>
<td>Correlation Subscription XML schema</td>
<td>80</td>
</tr>
<tr>
<td>8.41</td>
<td>Error attributes and model associations</td>
<td>82</td>
</tr>
<tr>
<td>8.42</td>
<td>Escalation attributes and model associations</td>
<td>83</td>
</tr>
<tr>
<td>8.43</td>
<td>FormalExpression attributes and model associations</td>
<td>86</td>
</tr>
<tr>
<td>8.44</td>
<td>FlowElement attributes and model associations</td>
<td>88</td>
</tr>
<tr>
<td>8.45</td>
<td>FlowElementsContainer model associations</td>
<td>89</td>
</tr>
<tr>
<td>8.46</td>
<td>Gateway attributes</td>
<td>91</td>
</tr>
<tr>
<td>8.47</td>
<td>ItemDefinition attributes &amp; model associations</td>
<td>92</td>
</tr>
<tr>
<td>8.48</td>
<td>Message attributes and model associations</td>
<td>95</td>
</tr>
<tr>
<td>8.49</td>
<td>Resource attributes and model associations</td>
<td>96</td>
</tr>
<tr>
<td>8.50</td>
<td>ResourceParameter attributes and model associations</td>
<td>97</td>
</tr>
<tr>
<td>8.51</td>
<td>SequenceFlow attributes and model associations</td>
<td>99</td>
</tr>
<tr>
<td>8.52</td>
<td>FlowNode model associations</td>
<td>100</td>
</tr>
<tr>
<td>8.53</td>
<td>Error XML schema</td>
<td>100</td>
</tr>
<tr>
<td>8.54</td>
<td>Escalation XML schema</td>
<td>100</td>
</tr>
<tr>
<td>8.55</td>
<td>Expression XML schema</td>
<td>100</td>
</tr>
<tr>
<td>8.56</td>
<td>FlowElement XML schema</td>
<td>101</td>
</tr>
<tr>
<td>8.57</td>
<td>FlowNode XML schema</td>
<td>101</td>
</tr>
<tr>
<td>8.58</td>
<td>FormalExpression XML schema</td>
<td>101</td>
</tr>
<tr>
<td>8.59</td>
<td>InputOutputBinding XML schema</td>
<td>102</td>
</tr>
<tr>
<td>8.60</td>
<td>ItemDefinition XML schema</td>
<td>102</td>
</tr>
<tr>
<td>8.61</td>
<td>Message XML schema</td>
<td>102</td>
</tr>
<tr>
<td>8.62</td>
<td>Resources XML schema</td>
<td>102</td>
</tr>
<tr>
<td>8.63</td>
<td>ResourceParameter XML schema</td>
<td>103</td>
</tr>
<tr>
<td>8.64</td>
<td>SequenceFlow XML schema</td>
<td>103</td>
</tr>
<tr>
<td>8.65</td>
<td>Interface attributes and model associations</td>
<td>105</td>
</tr>
<tr>
<td>8.66</td>
<td>Operation attributes and model associations</td>
<td>106</td>
</tr>
<tr>
<td>8.67</td>
<td>Interface XML schema</td>
<td>106</td>
</tr>
<tr>
<td>8.68</td>
<td>Operation XML schema</td>
<td>106</td>
</tr>
<tr>
<td>8.69</td>
<td>EndPoint XML schema</td>
<td>107</td>
</tr>
<tr>
<td>9.1</td>
<td>Collaboration Attributes and Model Associations</td>
<td>110</td>
</tr>
<tr>
<td>9.2</td>
<td>Participant attributes and model associations</td>
<td>116</td>
</tr>
<tr>
<td>9.3</td>
<td>PartnerEntity attributes</td>
<td>116</td>
</tr>
<tr>
<td>9.4</td>
<td>PartnerRole attributes</td>
<td>117</td>
</tr>
<tr>
<td>9.5</td>
<td>ParticipantMultiplicity attributes</td>
<td>118</td>
</tr>
<tr>
<td>9.6</td>
<td>ParticipantMultiplicity Instance attributes</td>
<td>118</td>
</tr>
<tr>
<td>9.7</td>
<td>ParticipantAssociation model associations</td>
<td>120</td>
</tr>
<tr>
<td>9.8</td>
<td>Message Flow attributes and model associations</td>
<td>123</td>
</tr>
<tr>
<td>9.9</td>
<td>MessageFlowAssociation attributes and model associations</td>
<td>124</td>
</tr>
<tr>
<td>9.10</td>
<td>ConversationNode Model Associations</td>
<td>130</td>
</tr>
<tr>
<td>9.11</td>
<td>Sub-Conversation Model Associations</td>
<td>131</td>
</tr>
<tr>
<td>9.12</td>
<td>Call Conversation Model Associations</td>
<td>132</td>
</tr>
</tbody>
</table>
Table 9.13  – Conversation Link Attributes and Model Associations 134
Table 9.14  – ConversationAssociation Model Associations 136
Table 9.15  – Call Conversation XML schema 139
Table 9.16  – Collaboration XML schema 140
Table 9.17  – Conversation XML schema 140
Table 9.18  – ConversationAssociation XML schema 140
Table 9.19  – ConversationAssociation XML schema 140
Table 9.20  – ConversationNode XML schema 141
Table 9.21  – Conversation Node XML schema 141
Table 9.22  – Global Conversation XML schema 141
Table 9.23  – MessageFlow XML schema 142
Table 9.24  – MessageFlowAssociation XML schema 142
Table 9.25  – Participant XML schema 142
Table 9.26  – ParticipantAssociation XML schema 143
Table 9.27  – ParticipantMultiplicity XML schema 143
Table 9.28  – PartnerEntity XML schema 143
Table 9.29  – PartnerRole XML schema 143
Table 9.30  – Sub-Conversation XML schema 144
Table 10.1  – Process Attributes & Model Associations 147
Table 10.2  – Process instance attributes 149
Table 10.3  – Activity attributes and model associations 152
Table 10.4  – Activity instance attributes 153
Table 10.5  – Resource Role model associations 155
Table 10.6  – ResourceAssignmentExpression model associations 155
Table 10.7  – ResourceParameterBinding model associations 156
Table 10.8  – Service Task model associations 159
Table 10.9  – Send Task model associations 161
Table 10.10  – Receive Task attributes and model associations 162
Table 10.11  – Business Rule Task attributes and model associations 164
Table 10.12  – Script Task attributes 165
Table 10.13  – User Task attributes and model associations 167
Table 10.14  – User Task instance attributes 167
Table 10.15  – ManualTask XML schema 168
Table 10.16  – UserTask XML schema 169
Table 10.17  – HumanPerformer XML schema 169
Table 10.18  – PotentialOwner XML schema 170
Table 10.19  – XML serialization of Buyer process 170
Table 10.20  – Sub-Process attributes 176
Table 10.21  – Transaction Sub-Process attributes and model associations 180
Table 10.22  – Ad-hoc Sub-Process model associations 181
Table 10.23  – CallActivity model associations 186
Table 10.24  – CallableElement attributes and model associations 187
Table 10.25  – InputOutputBinding model associations 187
Table 10.26  – Global Task model associations 188
Table 10.27  – Loop Activity instance attributes 190
Table 10.28  – StandardLoopCharacteristics attributes and model associations 191
Table 10.29  – MultiInstanceLoopCharacteristics attributes and model associations 192
Table 10.124 – InclusiveGateway Attributes & Model Associations 293
Table 10.125 – Complex Gateway model associations 296
Table 10.126 – Instance attributes related to the Complex Gateway 297
Table 10.127 – EventBasedGateway Attributes & Model Associations 300
Table 10.128 – ComplexGateway XML schema 301
Table 10.129 – EventBasedGateway XML schema 301
Table 10.130 – ExclusiveGateway XML schema 301
Table 10.131 – Gateway XML schema 301
Table 10.132 – InclusiveGateway XML schema 302
Table 10.133 – ParallelGateway XML schema 302
Table 10.134 – LaneSet attributes and model associations 308
Table 10.135 – Lane attributes and model associations 309
Table 10.136 – Process XML schema 312
Table 10.137 – Auditing XML schema 313
Table 10.138 – GlobalTask XML schema 313
Table 10.139 – Lane XML schema 313
Table 10.140 – LaneSet XML schema 313
Table 10.141 – Monitoring XML schema 314
Table 10.142 – Performer XML schema 314
Table 11.1 – Choreography Activity Model Associations 322
Table 11.2 – Choreography Task Model Associations 328
Table 11.3 – Sub-Choreography Model Associations 332
Table 11.4 – Call Choreography Model Associations 335
Table 11.5 – Global Choreography Task Model Associations 335
Table 11.6 – Use of Start Events in Choreography 340
Table 11.7 – Use of Intermediate Events in Choreography 341
Table 11.8 – Use of End Events in Choreography 343
Table 11.9 – Choreography XML schema 364
Table 11.10 – GlobalChoreographyTask XML schema 365
Table 11.11 – ChoreographyActivity XML schema 365
Table 11.12 – ChoreographyTask XML schema 365
Table 11.13 – CallChoreography XML schema 366
Table 11.14 – SubChoreography XML schema 366
Table 12.1 – BPMNDiagram XML schema 371
Table 12.2 – BPMNPlane XML schema 372
Table 12.3 – BPMNShape XML schema 374
Table 12.4 – BPMNEdge XML schema 376
Table 12.5 – BPMNLabel XML schema 377
Table 12.6 – BPMNLabelStyle XML schema 378
Table 12.7 – Complete BPMN DI XML schema 378
Table 12.8 – Depiction Resolution for Loop Compensation Marker 382
Table 12.9 – Depiction Resolution for Tasks 385
Table 12.10 – Depiction Resolution for Collapsed Sub-Processes 386
Table 12.11 – Depiction Resolution for Expanded Sub-Processes 386
Table 12.12 – Depiction Resolution for Collapsed Ad Hoc Sub-Processes 387
Table 12.13 – Depiction Resolution for Expanded Ad Hoc Sub-Processes 387
Table 12.14 – Depiction Resolution for Collapsed Transactions 387
Table 12.15 – Depiction Resolution for Tasks 388
Table 12.16 – Depiction Resolution for Collapsed Event Sub-Processes 388
Table 12.17 – Depiction Resolution for Expanded Event Sub-Processes 391
Table 12.18 – Depiction Resolution for Call Activities (Calling a Global Task) 391
Table 12.19 – Depiction Resolution for Collapsed Call Activities (Calling a Process) 392
Table 12.20 – Depiction Resolution for Expanded Call Activities (Calling a Process) 392
Table 12.21 – Depiction Resolution for Data 393
Table 12.22 – Depiction Resolution for Events 394
Table 12.23 – Depiction Resolution for Gateways 400
Table 12.24 – Depiction Resolution for Artifacts 401
Table 12.25 – Depiction Resolution for Lanes 401
Table 12.26 – Depiction Resolution for Pools 402
Table 12.27 – Depiction Resolution for Choreography Tasks 403
Table 12.28 – Depiction Resolution for Sub-Choreographies (Collapsed) 404
Table 12.29 – Depiction Resolution for Sub-Choreographies (Expanded) 405
Table 12.30 – Depiction Resolution for Call Choreographies (Calling a Global Choreography Task) 405
Table 12.31 – Depiction Resolution for Collapsed Call Choreographies (Calling a Choreography) 406
Table 12.32 – Depiction Resolution for Expanded Call Choreographies (Calling a Choreography) 407
Table 12.33 – Depiction Resolution for Choreography Participant Bands 408
Table 12.34 – Depiction Resolution for Conversations 410
Table 12.35 – Depiction Resolution for Connecting Objects 411
Table 12.36 – Expanded Sub-Process BPMN DI instance 413
Table 12.37 – Start and End Events on the Border BPMN DI instance 415
Table 12.38 – Collapsed Sub-Process BPMN DI instance 416
Table 12.39 – Sub-Process Content BPMN DI instance 417
Table 12.40 – Multiple Lanes and Nested Lanes BPMN DI instance 418
Table 12.41 – Vertical Collaboration BPMN DI instance 419
Table 12.42 – Conversation BPMN DI instance 420
Table 12.43 – Choreography BPMN DI instance 422
Table 13.1 – Parallel Gateway Execution Semantics 434
Table 13.2 – Exclusive Gateway Execution Semantics 435
Table 13.3 – Inclusive Gateway Execution Semantics 436
Table 13.4 – Event-Based Gateway Execution Semantics 437
Table 13.5 – Semantics of the Complex Gateway 438
Table 14.1 – Common Activity Mappings to WS-BPEL 448
Table 14.2 – Expressions mapping to WS-BPEL 468
Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at http://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications is available from the OMG website at:


Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM)
Platform Specific Model and Interface Specifications

- CORBA services
- CORBA facilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications

All of OMG’s formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.


Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Throughout the document, structural specifications will appear in paragraphs using a special shaped bullet: Example: ♦ A TASK MAY be a target for Sequence Flow; it can have multiple incoming Flows.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to http://www.omg.org/technology/agreement.htm.
1 Scope

The Object Management Group (OMG) has developed a standard Business Process Model and Notation (BPMN). The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation.

Another goal, but no less important, is to ensure that XML languages designed for the execution of business processes, such as WSBPEL (Web Services Business Process Execution Language), can be visualized with a business-oriented notation.

This specification represents the amalgamation of best practices within the business modeling community to define the notation and semantics of Collaboration diagrams, Process diagrams, and Choreography diagrams. The intent of BPMN is to standardize a business process model and notation in the face of many different modeling notations and viewpoints. In doing so, BPMN will provide a simple means of communicating process information to other business users, process implementers, customers, and suppliers.

The membership of the OMG has brought forth expertise and experience with many existing notations and has sought to consolidate the best ideas from these divergent notations into a single standard notation. Examples of other notations or methodologies that were reviewed are UML Activity Diagram, UML EDOC Business Processes, IDEF, ebXML BPSS, Activity-Decision Flow (ADF) Diagram, RosettaNet, LOVeM, and Event-Process Chains (EPCs).

2 Conformance

Software can claim compliance or conformance with BPMN 2.0 if and only if the software fully matches the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points can claim only that the software was based on this specification, but cannot claim compliance or conformance with this specification. The specification defines four types of conformance namely Process Modeling Conformance, Process Execution Conformance, BPEL Process Execution Conformance, and Choreography Modeling Conformance.

The implementation claiming conformance to Process Modeling Conformance type is NOT REQUIRED to support Choreography Modeling Conformance type and vice-versa. Similarly, the implementation claiming Process Execution Conformance type is NOT REQUIRED to be conformant to the Process Modeling and Choreography Conformance types.

The implementation claiming conformance to the Process Modeling Conformance type SHALL comply with all of the requirements set forth in Section 2.1. The implementation claiming conformance to the Process Execution Conformance type SHALL comply with all of the requirements set forth in Section 2.2. The implementation claiming conformance to the BPEL Process Execution Semantics Conformance type SHALL comply with all of the requirements set forth in Section 2.3. The implementation claiming conformance to the Choreography Conformance type SHALL comply with all of the requirements set forth in Section 2.4. The implementation is said to have BPMN Complete Conformance if it complies with all of the requirements stated in Sections 2.1, 2.2, 2.3, and 2.4.
2.1 Process Modeling Conformance

The next eight sections describe Process Modeling Conformance.

2.1.1 BPMN Process Types

The implementations claiming Process Modeling Conformance MUST support the following BPMN packages:

- The BPMN core elements, which include those defined in the Infrastructure, Foundation, Common, and Service packages (see Chapter 8).
- Process diagrams, which include the elements defined in the Process, Activities, Data, and Human Interaction packages (see Chapter 10).
- Collaboration diagrams, which include Pools and Message Flow (see Chapter 9).
- Conversation diagrams, which include Pools, Conversations, and Conversation Links (see Chapter 9).

As an alternative to full Process Modeling Conformance, there are three conformance sub-classes defined:

- Descriptive
- Analytic
- Common Executable

Descriptive is concerned with visible elements and attributes used in high-level modeling. It should be comfortable for analysts who have used BPA flowcharting tools.

Analytic contains all of Descriptive and in total about half of the constructs in the full Process Modeling Conformance Class. It is based on experience gathered in BPMN training and an analysis of user-patterns in the Department of Defense Architecture Framework and planned standardization for that framework.

Both Descriptive and Analytic focus on visible elements and a minimal subset of supporting attributes/elements.

Common Executable focuses on what is needed for executable process models.

Elements and attributes not in these sub-classes are contained in the full Process Modeling Conformance class.

The elements for each sub-class are defined in the next section.

2.1.2 BPMN Process Elements

The Process Modeling Conformance type set consists of Collaboration and Process diagram elements, including all Task types, embedded Sub-Processes, CallActivity, all Gateway types, all Event types (Start, Intermediate, and End), Lane, Participants, Data Object (including DatalInput and DataOutput), Message, Group, Text Annotation, Sequence Flow (including conditional and default flows), Message Flow, Conversations (limited to grouping Message Flow, and associating correlations), Correlation, and Association (including Compensation Association). The set also includes markers (Loop, Multi-Instance, Transaction, Compensation) for Tasks and embedded Sub-Processes.

Note: Implementations are not expected to support Choreography modeling elements such as Choreography Task and Sub-Choreography.

For a tool to claim support for a sub-class the following criteria MUST be satisfied:

- All the elements in the sub-class MUST be supported.
For each element, all the listed attributes MUST be supported.

In general, if the sub-class doesn’t mention an attribute and it is NOT REQUIRED by the schema, then it is not in the subclass. Exceptions to this rule are noted.

**Descriptive Conformance Sub-Class**

The Descriptive conformance sub-class elements are shown in Table 2.1.

**Table 2.1 – Descriptive Conformance Sub-Class Elements and Attributes**

<table>
<thead>
<tr>
<th>Element</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>participant (pool)</td>
<td>id, name, processRef</td>
</tr>
<tr>
<td>laneSet</td>
<td>id, lane with name, childLaneSet, flowElementRef</td>
</tr>
<tr>
<td>sequenceFlow (unconditional)</td>
<td>id, name, sourceRef, targetRef</td>
</tr>
<tr>
<td>messageFlow</td>
<td>id, name, sourceRef, targetRef</td>
</tr>
<tr>
<td>exclusiveGateway</td>
<td>id, name</td>
</tr>
<tr>
<td>parallelGateway</td>
<td>id, name</td>
</tr>
<tr>
<td>task (None)</td>
<td>id, name</td>
</tr>
<tr>
<td>userTask</td>
<td>id, name</td>
</tr>
<tr>
<td>serviceTask</td>
<td>id, name</td>
</tr>
<tr>
<td>subProcess (expanded)</td>
<td>id, name, flowElement</td>
</tr>
<tr>
<td>subProcess (collapsed)</td>
<td>id, name, flowElement</td>
</tr>
<tr>
<td>CallActivity</td>
<td>id, name, calledElement</td>
</tr>
<tr>
<td>DataObject</td>
<td>id, name</td>
</tr>
<tr>
<td>TextAnnotation</td>
<td>id, text</td>
</tr>
<tr>
<td>association/dataAssociationa</td>
<td>id, name, sourceRef, targetRef, associationDirectionb</td>
</tr>
<tr>
<td>dataStoreReference</td>
<td>id, name, dataStoreRef</td>
</tr>
<tr>
<td>startEvent (None)</td>
<td>id, name</td>
</tr>
<tr>
<td>endEvent (None)</td>
<td>id, name</td>
</tr>
<tr>
<td>messageStartEvent</td>
<td>id, name, messageEventDefinition</td>
</tr>
<tr>
<td>messageEndEvent</td>
<td>id, name, messageEventDefinition</td>
</tr>
<tr>
<td>timerStartEvent</td>
<td>id, name, timerEventDefinition</td>
</tr>
</tbody>
</table>
Analytic Conformance Sub-Class

The Analytic conformance sub-class contains all the elements of the Descriptive conformance sub-class plus the elements shown in Table 2.2.

Table 2.2 – Analytic Conformance Sub-Class Elements and Attributes

<table>
<thead>
<tr>
<th>Element</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminateEndEvent</td>
<td>id, name, terminateEventDefinition</td>
</tr>
<tr>
<td>documentation</td>
<td>text</td>
</tr>
<tr>
<td>Group</td>
<td>id, categoryRef</td>
</tr>
</tbody>
</table>

- **Data Association** is ABSTRACT: **Data Input Association** and **Data Output Association** will appear in the XML serialization. These both have REQUIRED attributes [sourceRef and targetRef] that refer to itemAwareElements. To be consistent with the metamodel, this will require the following additional elements: ioSpecification, inputSet, outputSet, **Data Input, Data Output**. When a **BPMN** editor draws a **Data Association** to an **Activity** or **Event** it should generate this supporting invisible substructure. Otherwise, the metamodel would have to be changed to make sourceRef and targetRef optional or allow reference to non-itemAwareElements, e.g., **Activity** and **Event**.
- associationDirection not specified for **Data Association**
- Documentation is not a visible element. It is an attribute of most elements.
<table>
<thead>
<tr>
<th>Event Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throwing message Intermediate Event</td>
<td>id, name, messageEventDefinition</td>
</tr>
<tr>
<td>Boundary message Intermediate Event</td>
<td>id, name, attachedToRef, messageEventDefinition</td>
</tr>
<tr>
<td>Non-interrupting Boundary message Intermediate Event</td>
<td>id, name, attachedToRef, cancelActivity=false, messageEventDefinition</td>
</tr>
<tr>
<td>Catching timer Intermediate Event</td>
<td>id, name, timerEventDefinition</td>
</tr>
<tr>
<td>Boundary timer Intermediate Event</td>
<td>id, name, attachedToRef, timerEventDefinition</td>
</tr>
<tr>
<td>Non-interrupting Boundary timer Intermediate Event</td>
<td>id, name, attachedToRef, cancelActivity=false, timerEventDefinition</td>
</tr>
<tr>
<td>Boundary error Intermediate Event</td>
<td>id, name, attachedToRef, errorEventDefinition</td>
</tr>
<tr>
<td>errorEndEvent</td>
<td>id, name, errorEventDefinition</td>
</tr>
<tr>
<td>Non-interrupting Boundary escalation Intermediate Event</td>
<td>id, name, attachedToRef, cancelActivity=false, escalationEventDefinition</td>
</tr>
<tr>
<td>Throwing escalation Intermediate Event</td>
<td>id, name, escalationEventDefinition</td>
</tr>
<tr>
<td>escalationEndEvent</td>
<td>id, name, escalationEventDefinition</td>
</tr>
<tr>
<td>Catching signal Intermediate Event</td>
<td>id, name, signalEventDefinition</td>
</tr>
<tr>
<td>Throwing signal Intermediate Event</td>
<td>id, name, signalEventDefinition</td>
</tr>
<tr>
<td>Boundary signal Intermediate Event</td>
<td>id, name, attachedToRef, signalEventDefinition</td>
</tr>
<tr>
<td>Non-interrupting Boundary signal Intermediate Event</td>
<td>id, name, attachedToRef, cancelActivity=false, signalEventDefinition</td>
</tr>
<tr>
<td>conditionalStartEvent</td>
<td>id, name, conditionalEventDefinition</td>
</tr>
<tr>
<td>Catching conditional Intermediate Event</td>
<td>id, name, conditionalEventDefinition</td>
</tr>
<tr>
<td>Boundary conditional Intermediate Event</td>
<td>id, name, conditionalEventDefinition</td>
</tr>
<tr>
<td>Non-interrupting Boundary conditional Intermediate Event</td>
<td>id, name, cancelActivity=false, conditionalEventDefinition</td>
</tr>
<tr>
<td>message</td>
<td>id, name, add messageRef attribute to messageFlow</td>
</tr>
</tbody>
</table>

a. **ConditionExpression**, allowed only for **Sequence Flow** out of **Gateways**, MAY be null.
b. **Default** is an attribute of a **sourceRef** (exclusive or inclusive) **Gateway**.
c. Note that **messageRef**, an attribute of various **Events**, is optional and not in the sub-class.
Common Executable Conformance Sub-Class

This conformance sub-class is intended for modeling tools that can emit executable models.

- Data type definition language MUST be XML Schema.
- Service Interface definition language MUST be WSDL.
- Data access language MUST be XPath.

The Common Executable conformance sub-class elements are shown in Table 2.3 and its supporting classes in Table 2.4.

Table 2.3 – Common Executable Conformance Sub-Class Elements and Attributes

<table>
<thead>
<tr>
<th>Element</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequenceFlow (unconditional)</td>
<td>id, (name), sourceRef&lt;sup&gt;a&lt;/sup&gt;, targetRef&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>sequenceFlow (conditional)</td>
<td>id, name, sourceRef, targetRef, conditionExpression&lt;sup&gt;c&lt;/sup&gt;</td>
</tr>
<tr>
<td>sequenceFlow (default)</td>
<td>id, name, sourceRef, targetRef, default&lt;sup&gt;d&lt;/sup&gt;</td>
</tr>
<tr>
<td>subProcess (expanded)</td>
<td>id, name, flowElement, loopCharacteristics, boundaryEventRefs</td>
</tr>
<tr>
<td>exclusiveGateway</td>
<td>id, name, gatewayDirection (only converging and diverging), default</td>
</tr>
<tr>
<td>parallelGateway</td>
<td>id, name, gatewayDirection (only converging and diverging)</td>
</tr>
<tr>
<td>startEvent (None)</td>
<td>id, name</td>
</tr>
<tr>
<td>endEvent (None)</td>
<td>id, name</td>
</tr>
<tr>
<td>eventBasedGateway</td>
<td>id, name, gatewayDirection, eventGatewayType</td>
</tr>
<tr>
<td>userTask</td>
<td>id, name, renderings, implementation, resources, ioSpecification, dataInputAssociations, dataOutputAssociations, loopCharacteristics, boundaryEventRefs</td>
</tr>
<tr>
<td>serviceTask</td>
<td>id, name, implementation, operationRef, ioSpecification, dataInputAssociations, dataOutputAssociations, loopCharacteristics, boundaryEventRefs</td>
</tr>
<tr>
<td>callActivity</td>
<td>id, name, calledElement, ioSpecification, dataInputAssociations, dataOutputAssociations, loopCharacteristics, boundaryEventRefs</td>
</tr>
<tr>
<td>dataObject</td>
<td>id, name, isCollection, itemSubjectRef</td>
</tr>
<tr>
<td>textAnnotation</td>
<td>id, text</td>
</tr>
<tr>
<td>dataAssociation</td>
<td>id, name, sourceRef, targetRef, assignment</td>
</tr>
<tr>
<td>messageStartEvent</td>
<td>id, name, messageEventDefinition (either ref or contained), dataOutput, dataOutputAssociations</td>
</tr>
<tr>
<td>messageEndEvent</td>
<td>id, name, messageEventDefinition, (either ref or contained), dataInput, dataInputAssociations</td>
</tr>
<tr>
<td>terminateEndEvent</td>
<td>(Terminating trigger in combination with one of the other end events)</td>
</tr>
</tbody>
</table>
### Table 2.3 – Common Executable Conformance Sub-Class Elements and Attributes

<table>
<thead>
<tr>
<th>Element</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catching message Intermediate Event</td>
<td>id, name, messageEventDefinition (either ref or contained), dataOutput, dataOutputAssociations</td>
</tr>
<tr>
<td>Throwing message Intermediate Event</td>
<td>id, name, messageEventDefinition (either ref or contained), dataInput, dataInputAssociations</td>
</tr>
<tr>
<td>Catching timer Intermediate Event</td>
<td>id, name, timerEventDefinition (contained)</td>
</tr>
<tr>
<td>Boundary error Intermediate Event</td>
<td>id, name, attachedToRef, errorEventDefinition, (contained or referenced), dataOutput, dataOutputAssociations</td>
</tr>
</tbody>
</table>

a. Multiple outgoing connections are only allowed for converging **Gateways**.
b. Multiple outgoing connections are only allowed for diverging **Gateways**.
c. **ConditionExpression**, allowed only for **Sequence Flow** out of **Gateways**, MAY be null.
d. **Default** is an attribute of a **sourceRef** (exclusive or inclusive) **Gateway**.

### Table 2.4 – Common Executable Conformance Sub-Class Supporting Classes

<table>
<thead>
<tr>
<th>Element</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>StandardLoopCharacteristics</td>
<td>id, loopCondition</td>
</tr>
<tr>
<td>MultiInstanceLoopCharacteristics</td>
<td>id, isSequential, loopDataInput, inputDataItem</td>
</tr>
<tr>
<td>Rendering</td>
<td></td>
</tr>
<tr>
<td>Resource</td>
<td>id, name</td>
</tr>
<tr>
<td>ResourceRole</td>
<td>id, resourceRef, resourceAssignmentExpression</td>
</tr>
<tr>
<td>InputOutputSpecification</td>
<td>id, dataInputs, dataOutputs</td>
</tr>
<tr>
<td>DataInput</td>
<td>id, name, isCollection, itemSubjectRef</td>
</tr>
<tr>
<td>DataOutput</td>
<td>id, name, isCollection, itemSubjectRef</td>
</tr>
<tr>
<td>ItemDefinition</td>
<td>id, structure or import&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>Operation</td>
<td>id, name, inMessageRef, outMessageRef, errorRefs</td>
</tr>
<tr>
<td>Message</td>
<td>id, name, structureRef</td>
</tr>
<tr>
<td>Error</td>
<td>id, structureRef</td>
</tr>
<tr>
<td>Assignment</td>
<td>id, from, to&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>MessageEventDefinition</td>
<td>id, messageRef, operationRef</td>
</tr>
<tr>
<td>TerminateEventDefinition</td>
<td>id</td>
</tr>
<tr>
<td>TimerEventDefinition</td>
<td>id, timeDate</td>
</tr>
</tbody>
</table>

<sup>a</sup> Structure MUST be defined by an XSD Complex Type

<sup>b</sup> Structure MUST be defined by an XSD Complex Type
2.1.3 Visual Appearance

A key element of BPMN is the choice of shapes and icons used for the graphical elements identified in this specification. The intent is to create a standard visual language that all process modelers will recognize and understand. An implementation that creates and displays BPMN Process Diagrams SHALL use the graphical elements, shapes, and markers illustrated in this specification.

Note – There is flexibility in the size, color, line style, and text positions of the defined graphical elements, except where otherwise specified (see page 41).

The following extensions to a BPMN Diagram are permitted:

- New markers or indicators MAY be added to the specified graphical elements. These markers or indicators could be used to highlight a specific attribute of a BPMN element or to represent a new subtype of the corresponding concept.
- A new shape representing a kind of Artifact MAY be added to a Diagram, but the new Artifact shape SHALL NOT conflict with the shape specified for any other BPMN element or marker.
- Graphical elements MAY be colored, and the coloring MAY have specified semantics that extend the information conveyed by the element as specified in this standard.
- The line style of a graphical element MAY be changed, but that change SHALL NOT conflict with any other line style REQUIRED by this specification.
- An extension SHALL NOT change the specified shape of a defined graphical element or marker (e.g., changing a square into a triangle, or changing rounded corners into squared corners, etc.).

2.1.4 Structural Conformance

An implementation that creates and displays BPMN diagrams SHALL conform to the specifications and restrictions with respect to the connections and other diagrammatic relationships between graphical elements. Where permitted or requested connections are specified as conditional and based on attributes of the corresponding concepts, the implementation SHALL ensure the correspondence between the connections and the values of those attributes.

Note – In general, these connections and relationships have specified semantic interpretations, which specify interactions among the process concepts represented by the graphical elements. Conditional relationships based on attributes represent specific variations in behavior. Structural conformance therefore guarantees the correct interpretation of the diagram as a specification of process, in terms of flows of control and information. Throughout the document, structural specifications will appear in paragraphs using a special shaped bullet: Example: A TASK MAY be a target for Sequence Flow; it can have multiple incoming Flows. An incoming Flow MAY be from an alternative path and/or parallel paths.

2.1.5 Process Semantics

This specification defines many semantic concepts used in defining Processes, and associates them with graphical elements, markers, and connections. To the extent that an implementation provides an interpretation of the BPMN diagram as a semantic specification of Process, the interpretation SHALL be consistent with the semantic interpretation herein specified. In other words, the implementation claiming BPMN Process Modeling Conformance has to support the semantics surrounding the diagram elements expressed in Chapter 10.

Note – The implementations claiming Process Modeling Conformance are not expected to support the BPMN execution semantics described in Chapter 13.
2.1.6 Attributes and Model Associations

This specification defines a number of attributes and properties of the semantic elements represented by the graphical elements, markers, and connections. Some of these attributes are purely representational and are so marked, and some have mandated representations. Some attributes are specified as mandatory, but have no representation or only optional representation. And some attributes are specified as optional. For every attribute or property that is specified as mandatory, a conforming implementation SHALL provide some mechanism by which values of that attribute or property can be created and displayed. This mechanism SHALL permit the user to create or view these values for each BPMN element specified to have that attribute or property. Where a graphical representation for that attribute or property is specified as REQUIRED, that graphical representation SHALL be used. Where a graphical representation for that attribute or property is specified as optional, the implementation MAY use either a graphical representation or some other mechanism. If a graphical representation is used, it SHALL be the representation specified. Where no graphical representation for that attribute or property is specified, the implementation MAY use either a graphical representation or some other mechanism. If a graphical representation is used, it SHALL NOT conflict with the specified graphical representation of any other BPMN element.

2.1.7 Extended and Optional Elements

A conforming implementation is NOT REQUIRED to support any element or attribute that is specified herein to be non-normative or informative. In each instance in which this specification defines a feature to be “optional,” it specifies whether the option is in:

- how the feature will be displayed
- whether the feature will be displayed
- whether the feature will be supported

A conforming implementation is NOT REQUIRED to support any feature whose support is specified to be optional. If an implementation supports an optional feature, it SHALL support it as specified. A conforming implementation SHALL support any “optional” feature for which the option is only in whether or how it SHALL be displayed.

2.1.8 Visual Interchange

One of the main goals of this specification is to provide an interchange format that can be used to exchange BPMN definitions (both domain model and diagram layout) between different tools. The implementation should support the metamodel for Process types specified in Section 13.1 to enable portability of process diagrams so that users can take business process definitions created in one vendor’s environment and use them in another vendor’s environment.

2.2 Process Execution Conformance

The next two sections describe Process Execution Conformance.
2.2.1 Execution Semantics

The BPMN execution semantics have been fully formalized in this version of the specification. The tool claiming BPMN Execution Conformance type MUST fully support and interpret the operational semantics and Activity life-cycle specified in Section 14.2.2. Non-operational elements listed in Chapter 14 MAY be ignored by implementations claiming BPMN Execution Conformance type. Conformant implementations MUST fully support and interpret the underlying metamodel.

Note – The tool claiming Process Execution Conformance type is not expected to support and interpret Choreography models. The tool claiming Process Execution Conformance type is not expected to support Process Modeling Conformance. More precisely, the tool is not required to support graphical syntax and semantics defined in this specification. It MAY use different graphical elements, shapes and markers, than those defined in this specification.

2.2.2 Import of Process Diagrams

The tool claiming Process Execution Conformance type MUST support import of BPMN Process diagram types including its definitional Collaboration (see Table 10.1).

2.3 BPEL Process Execution Conformance

Special type of Process Execution Conformance that supports the BPMN mapping to WS-BPEL as specified in Section 15.1 can claim BPEL Process Execution Conformance.

Note – The tool claiming BPEL Process Execution Conformance MUST fully support Process Execution Conformance. The tool claiming BPEL Process Execution Conformance is not expected to support and interpret Choreography models. The tool claiming BPEL Process Execution Conformance is not expected to support Process Modeling Conformance.

2.4 Choreography Modeling Conformance

The next five sections describe Choreography Conformance.

2.4.1 BPMN Choreography Types

The implementations claiming Choreography Conformance type MUST support the following BPMN packages:

- The BPMN core elements, which include those defined in the Infrastructure, Foundation, Common, and Service packages (see Chapter 8).
- Choreography diagrams, which includes the elements defined in the Choreography, and Choreography packages (see Chapter 11).
- Collaboration diagrams, which include Pools and Message Flow (see Chapter 9).

2.4.2 BPMN Choreography Elements

The Choreography Conformance set includes Message, Choreography Task, Global Choreography Task, Sub-Choreography (expanded and collapsed), certain types of Start Events (e.g., None, Timer, Conditional, Signal, and Multiple), certain types of Intermediate Events (None, Message attached to Activity boundary, Timer – normal as well as attached to Activity boundary, Timer used in Event Gateways, Cancel attached to an
Activity boundary, Conditional, Signal, Multiple, Link, etc.) and certain types of End Events (None and Terminate), and Gateways. In addition, to enable Choreography within Collaboration it should support Pools and Message Flow.

2.4.3 Visual Appearance

An implementation that creates and displays BPMN Choreography Diagrams SHALL use the graphical elements, shapes and markers as specified in the BPMN specification. The use of text, color, size, and lines for Choreography diagram types are listed in Section 7.4.

2.4.4 Choreography Semantics

The tool claiming Choreography Conformance should fully support and interpret the graphical and execution semantics surrounding Choreography diagram elements and Choreography diagram types.

2.4.5 Visual Interchange

The implementation should support import/export of Choreography diagram types and Collaboration diagram types that depict Choreography within collaboration as specified in Section 9.4 to enable portability of Choreography definitions, so that users can take BPMN definitions created in one vendor’s environment and use them in another vendor’s environment.
2.5 Summary of BPMN Conformance Types

Table 2.5 summarizes the requirements for BPMN Conformance.

<table>
<thead>
<tr>
<th>Category</th>
<th>Process Modeling Conformance</th>
<th>Process Execution Conformance</th>
<th>BPEL Process Execution Conformance</th>
<th>Choreography Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual representation of BPMN Diagram Types</td>
<td>Process diagram types and Collaboration diagram types depicting collaborations among Process diagram types.</td>
<td>N/A</td>
<td>N/A</td>
<td>Choreography diagram types and Collaboration diagram types depicting collaboration among Choreography diagram types.</td>
</tr>
<tr>
<td>BPMN Diagram Elements that need to be supported.</td>
<td>All Task types, embedded Sub-Process, Call Activity, all Event types, all Gateway types, Pool, Lane, Data Object (including DataInput and DataOutput), Message, Group, Artifacts, markers for Tasks and Sub-Processes, Sequence Flow, Associations, and Message Flow.</td>
<td>N/A</td>
<td>N/A</td>
<td>Message, Choreography Task, Global Choreography Task, Sub-Choreography (expanded and collapsed), certain types of Start, Intermediate, and End Events, Gateways, Pools and Message Flow.</td>
</tr>
<tr>
<td>Support for Graphical syntax and semantics</td>
<td>Process and Collaboration diagrams that depict Process within Collaboration.</td>
<td>N/A</td>
<td>N/A</td>
<td>Choreography and Collaboration diagrams depicting Choreography within Collaboration.</td>
</tr>
<tr>
<td>Support for Execution Semantics</td>
<td>N/A</td>
<td>Yes for Process diagrams</td>
<td>Yes for Process diagrams</td>
<td>Choreography execution semantics</td>
</tr>
</tbody>
</table>

3 Normative References

3.1 Normative

RFC-2119

- Key words for use in RFCs to Indicate Requirement Levels, S. Bradner, IETF RFC 2119, March 1997
  http://www.ietf.org/rfc/rfc2119.txt
3.2 Non-Normative

Activity Service

- Additional Structuring Mechanism for the OTS specification, OMG, June 1999
  http://www.omg.org
- J2EE Activity Service for Extended Transactions (JSR 95), JCP
  http://www.jcp.org/jsr/detail/95.jsp

BPEL4People

- WS-BPEL Extension for People (BPEL4People) 1.0, June 2007
  http://www.active-endpoints.com/active-bpel-for-people.htm
- http://dev2dev.bea.com/arch2arch/
- https://www.sdn.sap.com/irj/sdn/bpel4people

Business Process Definition Metamodel

- OMG, May 2008,

Business Process Modeling

  http://www.ebpml.org/ebpml2.2.doc

Business Transaction Protocol

- OASIS BTP Technical Committee, June, 2002

Dublin Core Meta Data

- Dublin Core Metadata Element Set, Dublin Core Metadata Initiative
  http://dublincore.org/documents/dces/

ebXML BPSS

  http://www.ebpml.org/ebpml.doc
OMG UML

- Unified Modeling Language Specification V2.1.2: Superstructure, OMG, Nov 2007,
  http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF

Open Nested Transactions

- Concepts and Applications of Multilevel Transactions and Open Nested Transactions, Gerhard Weikum, Hans-J. Schek, 1992
  http://citeseer.nj.nec.com/weikum92concepts.html

RDF

- RDF Vocabulary Description Language 1.0: RDF Schema, W3C Working Draft
  http://www.w3.org/TR/rdf-schema/

SOAP 1.2

  http://www.w3.org/TR/soap12-part1/
- SOAP Version 1.2 Part 2: Adjuncts, W3C Working Draft
  http://www.w3.org/TR/soap12-part2/

UDDI

- Universal Description, Discovery and Integration, Ariba, IBM and Microsoft, UDDI.org.
  http://www.uddi.org

URI

  http://www.ietf.org/rfc/rfc2396.txt

WfMC Glossary

- Workflow Management Coalition Terminology and Glossary

Web Services Transaction

- (WS-Transaction) 1.1, OASIS, 12 July 2007,
  http://www.oasis-open.org/committees/ws-tx/

Workflow Patterns

  http://www.workflowpatterns.com/
WSBPEL

- Web Services Business Process Execution Language (WSBPEL) 2.0, OASIS Standard, April 2007  
  http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

WS-Coordination

- Web Services Coordination (WS-Coordination) 1.1, OASIS Standard, July 2007  
  http://www.oasis-open.org/committees/ws-tx/

WSDL

- Web Services Description Language (WSDL) 2.0, W3C Proposed Recommendation, June 2007  
  http://www.w3.org/TR/wsdl20/

WS-HumanTask

- Web Services Human Task (WS-HumanTask) 1.0, June 2007  
  http://www.active-endpoints.com/active-bpel-for-people.htm
- http://dev2dev.bea.com/arch2arch/
- https://www.sdn.sap.com/irj/sdn/bpel4people

XML 1.0 (Second Edition)

  http://www.w3.org/TR/REC-xml

XML-Namespace

- Namespaces in XML, Tim Bray et al., eds., W3C, 14 January 1999  
  http://www.w3.org/TR/REC-xml-names

XML-Schema

  http://www.w3.org/TR/xmlschema-1/
  http://www.w3.org/TR/xmlschema-2/

XPath

- XML Path Language (XPath) 1.0, James Clark and Steve DeRose, eds., W3C, 16 November 1999  
  http://www.w3.org/TR/xpath
4 Terms and Definitions

See Annex C - Glossary.

5 Symbols

There are no symbols defined in this specification.

6 Additional Information

6.1 Conventions

The section introduces the conventions used in this document. This includes (text) notational conventions and notations for schema components. Also included are designated namespace definitions.

6.1.1 Typographical and Linguistic Conventions and Style

This specification incorporates the following conventions:

- The keywords “MUST,” “MUST NOT,” “REQUIRED,” “SHALL,” “MUST NOT,” “SHOULD,” “SHOULD NOT,” “RECOMMENDED,” “MAY,” and “OPTIONAL” in this document are to be interpreted as described in RFC-2119.

- A term is a word or phrase that has a special meaning. When a term is defined, the term name is highlighted in bold typeface.

- A reference to another definition, section, or specification is highlighted with underlined typeface and provides a link to the relevant location in this specification.

- A reference to a graphical element is highlighted with a bold, capitalized word and will be presented with the Arial font (e.g., Sub-Process).

- A reference to a non-graphical element or BPMN concept is highlighted by being italicized and will be presented with the Times New Roman font (e.g., token).

- A reference to an attribute or model association will be presented with the Courier New font (e.g., Expression).

- A reference to a WSBPEL element, attribute, or construct is highlighted with an italic lower-case word, usually preceded by the word “WSBPEL” and will be presented with the Courier New font (e.g., WSBPEL pick).

- Non-normative examples are set off in boxes and accompanied by a brief explanation.
• XML and pseudo code is highlighted with mono-spaced typeface. Different font colors MAY be used to highlight the different components of the XML code.

• The cardinality of any content part is specified using the following operators:
  • <none> — exactly once
  • [0..1] — 0 or 1
  • [0..*] — 0 or more
  • [1..*] — 1 or more
• Attributes separated by | and grouped within { and } — alternative values
  • <value> — default value
  • <type> — the type of the attribute

6.1.2 Abbreviations

The following abbreviations are used throughout this document:

<table>
<thead>
<tr>
<th>This abbreviation</th>
<th>Refers to</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSBPEL</td>
<td>Web Services Business Process Execution Language (see WSBPEL). This abbreviation refers specifically to version 2.0 of the specification.</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Service Description Language (see WSDL). This abbreviation refers specifically to the W3C Technical Note, 15 March 2001, but is intended to support future versions of the WSDL specification.</td>
</tr>
</tbody>
</table>

6.2 Structure of this Document

Chapter 7 discusses the scope of the specification and provides a summary of the elements introduced in subsequent sections of the document.

Chapter 8 introduces the BPMN Core that includes basic BPMN elements needed for constructing various Business Processes, including collaborations, orchestration Processes and Choreographies.

Elements needed for modeling of Collaborations, orchestration Processes, Conversations, and Choreographies are introduced in chapters 9, 10, 11 and 12, respectively.

Chapter 13 introduces the BPMN visual diagram model. Chapter 14 defines the execution semantics for Process orchestrations in BPMN 2.0. Chapter 15 discusses a mapping of a BPMN model to WS-BPEL that is derived by analyzing the BPMN objects and the relationships between these objects. Exchange formats and an XSLT transformation between them are provided in Chapter 16.

6.3 Acknowledgements

Submitting Organizations (RFP Process)

The following companies are formal submitting members of OMG:

• Axway
• International Business Machines
• MEGA International
• Oracle
• SAP AG
• Unisys

<table>
<thead>
<tr>
<th>Supporting Organizations (RFP Process)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following organizations support this specification but are not formal submitters:</td>
</tr>
<tr>
<td>• Accenture</td>
</tr>
<tr>
<td>• Adaptive</td>
</tr>
<tr>
<td>• BizAgi</td>
</tr>
<tr>
<td>• Bruce Silver Associates</td>
</tr>
<tr>
<td>• Capgemini</td>
</tr>
<tr>
<td>• Enterprise Agility</td>
</tr>
<tr>
<td>• France Telecom</td>
</tr>
<tr>
<td>• IDS Scheer</td>
</tr>
<tr>
<td>• Intalio</td>
</tr>
<tr>
<td>• Metastorm</td>
</tr>
<tr>
<td>• Model Driven Solutions</td>
</tr>
<tr>
<td>• Nortel</td>
</tr>
<tr>
<td>• Red Hat Software</td>
</tr>
<tr>
<td>• Software AG</td>
</tr>
<tr>
<td>• TIBCO Software</td>
</tr>
<tr>
<td>• Vangent</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Finalization Task Force Voting Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following organizations have been Voting Members of the BPMN 2.0 Finalization Task Force:</td>
</tr>
<tr>
<td>• Adaptive</td>
</tr>
<tr>
<td>• Axway Software</td>
</tr>
<tr>
<td>• BAE SYSTEMS</td>
</tr>
<tr>
<td>• BizAgi Ltd.</td>
</tr>
<tr>
<td>• CA Inc.</td>
</tr>
<tr>
<td>• Camunda Services GmbH</td>
</tr>
<tr>
<td>• Cordys</td>
</tr>
<tr>
<td>• DICOM</td>
</tr>
<tr>
<td>• France Telecom R&amp;D</td>
</tr>
<tr>
<td>• Fujitsu</td>
</tr>
<tr>
<td>• Global 360, Inc.</td>
</tr>
</tbody>
</table>
• Hewlett-Packard
• iGrafx
• Inferware
• Intalio
• International Business Machines
• KnowGravity Inc.
• Lombardi Software
• MITRE
• U.S. National Institute of Standards and Technology
• No Magic, Inc.
• ooze Innovative Informatik GmbH
• Oracle
• PNA Group
• Red Hat
• SAP AG
• Softeam
• Software AG Inc.
• TIBCO
• Trisotech
• Visumpoint

Special Acknowledgements

The following persons were members of the core teams that contributed to the content of this specification: Anurag Aggarwal, Mike Amend, Sylvain Astier, Alistair Barros, Rob Bartel, Mariano Benitez, Conrad Bock, Gary Brown, Justin Brunt, John Bulles, Martin Chapman, Fred Cummins, Rouven Day, Maged Elaasar, David Frankel, Denis Gagné, John Hall, Reiner Hille-Doering, Dave Ings, Pablo Irassar, Oliver Kieselbach, Matthias Kloppmann, Jana Koehler, Frank Michael Kraft, Tammo van Lessen, Frank Leymann, Antoine Lonjon, Sumeet Malhotra, Falko Menge, Jeff Mischkinsky, Dale Moberg, Alex Moffat, Ralf Mueller, Sjir Nijssen, Karsten Ploesser, Pete Rivett, Michael Rowley, Bernd Ruecker, Tom Rutt, Suzette Samoojh, Robert Shapiro, Vishal Saxena, Scott Schanel, Axel Scheithauer, Bruce Silver, Meera Srinivasan, Antoine Toulme, Ivana Trickovic, Hagen Voelzer, Franz Weber, Andrea Westerinen and Stephen A. White.

In addition, the following persons contributed valuable ideas and feedback that improved the content and the quality of this specification: im Amsden, Mariano Belaunde, Peter Carlson, Cory Casanave, Michele Chinosi, Manoj Das, Robert Lario, Sumeet Malhotra, Henk de Man, David Marston, Neal McWhorter, Edita Mileviciene, Vadim Pevzner, Pete Rivett, Jesus Sanchez, Markus Schacher, Sebastian Stein, and Prasad Yendluri.
7 Overview

There has been much activity in the past few years in developing web service-based XML execution languages for Business Process Management (BPM) systems. Languages such as WSBPEL provide a formal mechanism for the definition of business processes. The key element of such languages is that they are optimized for the operation and inter-operation of BPM Systems. The optimization of these languages for software operations renders them less suited for direct use by humans to design, manage, and monitor Business Processes. WSBPEL has both graph and block structures and utilizes the principles of formal mathematical models, such as pi-calculus\(^1\). This technical underpinning provides the foundation for business process execution to handle the complex nature of both internal and B2B interactions and take advantage of the benefits of Web services. Given the nature of WSBPEL, a complex Business Process could be organized in a potentially complex, disjointed, and unintuitive format that is handled very well by a software system (or a computer programmer), but would be hard to understand by the business analysts and managers tasked to develop, manage, and monitor the Process. Thus, there is a human level of “inter-operability” or “portability” that is not addressed by these web service-based XML execution languages.

Business people are very comfortable with visualizing Business Processes in a flow-chart format. There are thousands of business analysts studying the way companies work and defining Business Processes with simple flow charts. This creates a technical gap between the format of the initial design of Business Processes and the format of the languages, such as WSBPEL, that will execute these Business Processes. This gap needs to be bridged with a formal mechanism that maps the appropriate visualization of the Business Processes (a notation) to the appropriate execution format (a BPM execution language) for these Business Processes.

Inter-operation of Business Processes at the human level, rather than the software engine level, can be solved with standardization of the Business Process Model and Notation (BPMN). BPMN provides a multiple diagrams, which are designed for use by the people who design and manage Business Processes. BPMN also provides a mapping to an execution language of BPM Systems (WSBPEL). Thus, BPMN would provide a standard visualization mechanism for Business Processes defined in an execution optimized business process language.

BPMN provides businesses with the capability of understanding their internal business procedures in a graphical notation and will give organizations the ability to communicate these procedures in a standard manner. Currently, there are scores of Process modeling tools and methodologies. Given that individuals will move from one company to another and that companies will merge and diverge, it is likely that business analysts need to understand multiple representations of Business Processes—potentially different representations of the same Process as it moves through its lifecycle of development, implementation, execution, monitoring, and analysis. Therefore, a standard graphical notation will facilitate the understanding of the performance Collaborations and business transactions within and between the organizations. This will ensure that businesses will understand themselves and participants in their business and will enable organizations to adjust to new internal and B2B business circumstances quickly. BPMN follows the tradition of flowcharting notations for readability and flexibility. In addition, the BPMN execution semantics is fully formalized. The OMG is using the experience of the business process notations that have preceded BPMN to create the next generation notation that combines readability, flexibility, and expandability.

BPMN will also advance the capabilities of traditional business process notations by inherently handling B2B Business Process concepts, such as public and private Processes and Choreographies, as well as advanced modeling concepts, such as exception handling, transactions, and compensation.

---

7.1 BPMN Scope

This specification provides a notation and model for Business Processes and an interchange format that can be used to exchange BPMN Process definitions (both domain model and diagram layout) between different tools. The goal of the specification is to enable portability of Process definitions, so that users can take Process definitions created in one vendor’s environment and use them in another vendor’s environment.

The BPMN 2.0 specification extends the scope and capabilities of the BPMN 1.2 in several areas:

- Formalizes the execution semantics for all BPMN elements
- Defines an extensibility mechanism for both Process model extensions and graphical extensions
- Refines Event composition and correlation
- Extends the definition of human interactions
- Defines a Choreography model

This specification also resolves known BPMN 1.2 inconsistencies and ambiguities.

BPMN is constrained to support only the concepts of modeling that are applicable to Business Processes. This means that other types of modeling done by organizations for business purposes is out of scope for BPMN. Therefore, the following are aspects that are out of the scope of this specification:

- Definition of organizational models and resources
- Modeling of functional breakdowns
- Data and information models
- Modeling of strategy
- Business rules models

Since these types of high-level modeling either directly or indirectly affects Business Processes, the relationships between BPMN and other high-level business modeling can be defined more formally as BPMN and other specifications are advanced.

While BPMN shows the flow of data (Messages), and the association of data artifacts to Activities, it is not a data flow language. In addition, operational simulation, monitoring and deployment of Business Processes are out of scope of this specification.

BPMN 2.0 can be mapped to more than one platform dependent process modeling language, e.g., WS-BPEL 2.0. This document includes a mapping of a subset of BPMN to WS-BPEL 2.0. Mappings to other emerging standards are considered to be separate efforts.

The specification utilizes other standards for defining data types, Expressions and service operations. These standards are XML Schema, XPath, and WSDL, respectively.

7.1.1 Uses of BPMN

Business Process modeling is used to communicate a wide variety of information to a wide variety of audiences. BPMN is designed to cover many types of modeling and allows the creation of end-to-end Business Processes. The structural elements of BPMN allow the viewer to be able to easily differentiate between sections of a BPMN Diagram. There are three basic types of sub-models within an end-to-end BPMN model:
1. **Processes** (*Orchestration*), including:
   - *Private non-executable* (internal) **Business Processes**
   - *Private executable* (internal) **Business Processes**
   - *Public Processes*

2. **Choreographies**

3. **Collaborations**, which can include **Processes** and/or **Choreographies**
   - A view of **Conversations**

### Private (Internal) Business Processes

*Private Business Processes* are those internal to a specific organization. These **Processes** have been generally called workflow or **BPM Processes** (see Figure 10.4). Another synonym typically used in the Web services area is the *Orchestration* of services. There are two types of *private Processes*: *executable* and *non-executable*. An *executable Process* is a Process that has been modeled for the purpose of being executed according to the semantics defined in Chapter 14. Of course, during the development cycle of the **Process**, there will be stages where the **Process** does not have enough detail to be “executable.” A non-executable **Process** is a *private Process* that has been modeled for the purpose of documenting **Process** behavior at a modeler-defined level of detail. Thus, information needed for execution, such as formal condition Expressions are typically not included in a *non-executable Process*.

If a swimlanes-like notation is used (e.g., a **Collaboration**, see below) then a *private Business Process* will be contained within a single **Pool**. The **Process** flow is therefore contained within the **Pool** and cannot cross the boundaries of the **Pool**. The flow of **Messages** can cross the **Pool** boundary to show the interactions that exist between separate *private Business Processes*.

![Diagram](image.png)

**Figure 7.1 - Example of a private Business Process**

### Public Processes

A **public Process** represents the interactions between a *private Business Process* and another **Process** or **Participant** (see Figure 7.2). Only those **Activities** that are used to communicate to the other **Participant(s)** are included in the **public Process**. All other “internal” **Activities** of the **private Business Process** are not shown in the **public Process**. Thus, the **public Process** shows to the outside world the **Message Flows** and the order of those **Message Flows** that are needed to interact with that **Process**. **Public Processes** can be modeled separately or within a **Collaboration** to show the flow of **Messages** between the **public Process Activities** and other **Participants**. Note that the **public** type of **Process** was named “abstract” in **BPMN 1.2**.
A **Collaboration** depicts the interactions between two or more business entities. A **Collaboration** usually contains two or more **Pools**, representing the **Participants** in the **Collaboration**. The **Message** exchange between the **Participants** is shown by a **Message Flow** that connects two **Pools** (or the objects within the **Pools**). The **Messages** associated with the **Message Flows** can also be shown. The **Collaboration** can be shown as two or more **public Processes** communicating with each other (see Figure 7.3). With a **public Process**, the **Activities** for the **Collaboration** participants can be considered the “touch-points” between the participants. The corresponding internal (executable) **Processes** are likely to have much more **Activity** and detail than what is shown in the **public Processes**. Or a **Pool** MAY be empty, a “black box.” **Choreographies** MAY be shown “in between” the **Pools** as they bisect the **Message Flows** between the **Pools**. All combinations of **Pools, Processes**, and a **Choreography** are allowed in a **Collaboration**.

---

**Figure 7.2 - Example of a public Process**

---

**Collaborations**
Figure 7.3 - An example of a Collaborative Process

Choreographies

A self-contained Choreography (no Pools or Orchestration) is a definition of the expected behavior, basically a procedural contract, between interacting Participants. While a normal Process exists within a Pool, a Choreography exists between Pools (or Participants).

The Choreography looks similar to a private Business Process since it consists of a network of Activities, Events, and Gateways (see Figure 7.4). However, a Choreography is different in that the Activities are interactions that represent a set (1 or more) of Message exchanges, which involves two or more Participants. In addition, unlike a normal Process, there is no central controller, responsible entity or observer of the Process.
Conversations

The Conversation diagram is a particular usage of and an informal description of a Collaboration diagram. However, the Pools of a Conversation usually do not contain a Process and a Choreography is usually not placed in between the Pools of a Conversation diagram. A Conversation is the logical relation of Message exchanges. The logical relation, in practice, often concerns a business object(s) of interest, e.g., “Order,” “Shipment and Delivery,” or “Invoice.”

Message exchanges are related to each other and reflect distinct business scenarios. For example, in logistics, stock replenishments involve the following type scenarios: creation of sales orders; assignment of carriers for shipments combining different sales orders; crossing customs/quarantine; processing payment and investigating exceptions. Thus, a Conversation diagram, as shown in Figure 7.5, shows Conversations (as hexagons) between Participants (Pools). This provides a “bird’s eye” perspective of the different Conversations that relate to the domain.

Figure 7.5 - An example of a Conversation diagram

Diagram Point of View

Since a BPMN Diagram MAY depict the Processes of different Participants, each Participant could view the Diagram differently. That is, the Participants have different points of view regarding how the Processes will apply to them. Some of the Activities will be internal to the Participant (meaning performed by or under control of the Participant) and other
Activities will be external to the Participant. Each Participant will have a different perspective as to which are internal and external. At runtime, the difference between internal and external Activities is important in how a Participant can view the status of the Activities or troubleshoot any problems. However, the Diagram itself remains the same. Figure 7.3 displays a Business Process that has two points of view. One point of view is of a Patient, the other is of the Doctor’s office. The Diagram shows the Activities of both participants in the Process, but when the Process is actually being performed, each Participant will only have control over their own Activities. Although the Diagram point of view is important for a viewer of the Diagram to understand how the behavior of the Process will relate to that viewer, BPMN will not currently specify any graphical mechanisms to highlight the point of view. It is open to the modeler or modeling tool vendor to provide any visual cues to emphasize this characteristic of a Diagram.

Understanding the Behavior of Diagrams

Throughout this document, we discuss how Sequence Flows are used within a Process. To facilitate this discussion, 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.

A Start Event generates a token that MUST eventually be consumed at an End Event (which MAY be implicit if not graphically displayed). The path of tokens should be traceable through the network of Sequence Flows, Gateways, and Activities within a Process.

Note – A token does not traverse a Message Flow since it is a Message that is passed down a Message Flow (as the name implies).

7.2 BPMN Elements

It should be emphasized that one of the drivers for the development of BPMN is to create a simple and understandable mechanism for creating Business Process models, while at the same time being able to handle the complexity inherent to Business Processes. The approach taken to handle these two conflicting requirements was to organize the graphical aspects of the notation into specific categories. This provides a small set of notation categories so that the reader of a BPMN diagram can easily recognize the basic types of elements and understand the diagram. Within the basic categories of elements, additional variation and information can be added to support the requirements for complexity without dramatically changing the basic look and feel of the diagram. The five basic categories of elements are:

1. Flow Objects
2. Data
3. Connecting Objects
4. Swimlanes
5. Artifacts

Flow Objects are the main graphical elements to define the behavior of a Business Process. There are three Flow Objects:

1. Events
2. Activities
3. Gateways
Data is represented with the four elements:

1. Data Objects
2. Data Inputs
3. Data Outputs
4. Data Stores

There are four ways of connecting the Flow Objects to each other or other information. There are four Connecting Objects:

1. Sequence Flows
2. Message Flows
3. Associations
4. Data Associations

There are two ways of grouping the primary modeling elements through “Swimlanes:”

1. Pools
2. Lanes

Artifacts are used to provide additional information about the Process. There are two standardized Artifacts, but modelers or modeling tools are free to add as many Artifacts as necessary. There could be additional BPMN efforts to standardize a larger set of Artifacts for general use or for vertical markets. The current set of Artifacts includes:

- Group
- Text Annotation

7.2.1 Basic BPMN Modeling Elements

Table 7.1 displays a list of the basic modeling elements that are depicted by the notation.
Table 7.1 - Basic Modeling Elements

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>An Event is something that “happens” during the course of a Process (see page 238) or a Choreography (see page 339). These Events affect the flow of the model and usually have a cause (trigger) or an impact (result). Events are circles with open centers to allow internal markers to differentiate different triggers or results. There are three types of Events, based on when they affect the flow: Start, Intermediate, and End.</td>
<td><img src="image" alt="Event" /></td>
</tr>
<tr>
<td>Activity</td>
<td>An Activity is a generic term for work that company performs (see page 151) in a Process. An Activity can be atomic or non-atomic (compound). The types of Activities that are a part of a Process Model are: Sub-Process and Task, which are rounded rectangles. Activities are used in both standard Processes and in Choreographies.</td>
<td><img src="image" alt="Activity" /></td>
</tr>
<tr>
<td>Gateway</td>
<td>A Gateway is used to control the divergence and convergence of Sequence Flows in a Process (see page 145) and in a Choreography (see page 344). Thus, it will determine branching, forking, merging, and joining of paths. Internal markers will indicate the type of behavior control.</td>
<td><img src="image" alt="Gateway" /></td>
</tr>
<tr>
<td>Sequence Flow</td>
<td>A Sequence Flow is used to show the order that Activities will be performed in a Process (see page 97) and in a Choreography (see page 320).</td>
<td><img src="image" alt="Sequence Flow" /></td>
</tr>
<tr>
<td>Message Flow</td>
<td>A Message Flow is used to show the flow of Messages between two Participants that are prepared to send and receive them (see page 120). In BPMN, two separate Pools in a Collaboration Diagram will represent the two Participants (e.g., PartnerEntities and/or PartnerRoles).</td>
<td><img src="image" alt="Message Flow" /></td>
</tr>
<tr>
<td>Association</td>
<td>An Association is used to link information and Artifacts with BPMN graphical elements (see page 67). Text Annotations (see page 71) and other Artifacts (see page 66) can be Associated with the graphical elements. An arrowhead on the Association indicates a direction of flow (e.g., data), when appropriate.</td>
<td><img src="image" alt="Association" /></td>
</tr>
</tbody>
</table>
### Table 7.1 - Basic Modeling Elements

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pool</td>
<td>A Pool is the graphical representation of a Participant in a Collaboration (see page 112). It also acts as a &quot;swimlane&quot; and a graphical container for partitioning a set of Activities from other Pools, usually in the context of B2B situations. A Pool MAY have internal details, in the form of the Process that will be executed. Or a Pool MAY have no internal details, i.e., it can be a &quot;black box.&quot;</td>
</tr>
<tr>
<td>Lane</td>
<td>A Lane is a sub-partition within a Process, sometimes within a Pool, and will extend the entire length of the Process, either vertically or horizontally (see on page 305). Lanes are used to organize and categorize Activities.</td>
</tr>
<tr>
<td>Data Object</td>
<td>Data Objects provide information about what Activities require to be performed and/or what they produce (see page 205), Data Objects can represent a singular object or a collection of objects. Data Input and Data Output provide the same information for Processes.</td>
</tr>
<tr>
<td>Message</td>
<td>A Message is used to depict the contents of a communication between two Participants (as defined by a business PartnerRole or a business PartnerEntity—see on page 93).</td>
</tr>
<tr>
<td>Group (a box around a group of objects within the same category)</td>
<td>A Group is a grouping of graphical elements that are within the same Category (see page 70). This type of grouping does not affect the Sequence Flows within the Group. The Category name appears on the diagram as the group label. Categories can be used for documentation or analysis purposes. Groups are one way in which Categories of objects can be visually displayed on the diagram.</td>
</tr>
<tr>
<td>Text Annotation (attached with an Association)</td>
<td>Text Annotations are a mechanism for a modeler to provide additional text information for the reader of a BPMN Diagram (see page 71).</td>
</tr>
</tbody>
</table>

### 7.2.2 Extended BPMN Modeling Elements

Table 7.2 displays a more extensive list of the Business Process concepts that could be depicted through a business process modeling notation.
<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>An Event is something that “happens” during the course of a Process (see page 238) or a Choreography (see page 339). These Events affect the flow of the model and usually have a cause (Trigger) or an impact (Result). Events are circles with open centers to allow internal markers to differentiate different Triggers or Results. There are three types of Events, based on when they affect the flow: Start, Intermediate, and End.</td>
<td></td>
</tr>
<tr>
<td>Flow Dimension (e.g., Start, Intermediate, End)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start</td>
<td>As the name implies, the Start Event indicates where a particular Process (see page 238) or Choreography (see page 339) will start.</td>
<td></td>
</tr>
<tr>
<td>Intermediate</td>
<td>Intermediate Events occur between a Start Event and an End Event. They will affect the flow of the Process (see page 249) or Choreography (see page 341), but will not start or (directly) terminate the Process.</td>
<td></td>
</tr>
<tr>
<td>End</td>
<td>As the name implies, the End Event indicates where a Process (see page 246) or Choreography (see page 343) will end.</td>
<td></td>
</tr>
</tbody>
</table>
The Start and some Intermediate Events have “triggers” that define the cause for the Event (see section entitled “Start Event” on page 238 and section entitled “Intermediate Event” on page 249). There are multiple ways that these events can be triggered. End Events MAY define a “result” that is a consequence of a Sequence Flow path ending. Start Events can only react to (“catch”) a trigger. End Events can only create (“throw”) a result. Intermediate Events can catch or throw triggers. For the Events, triggers that catch, the markers are unfilled, and for triggers and results that throw, the markers are filled.

Additionally, some Events, which were used to interrupt Activities in BPMN 1.1, can now be used in a mode that does not interrupt. The boundary of these Events is dashed (see figure to the right).

### Table 7.2 - BPMN Extended Modeling Elements

<table>
<thead>
<tr>
<th>Type Dimension (e.g., None, Message, Timer, Error, Cancel, Compensation, Conditional, Link, Signal, Multiple, Terminate.)</th>
<th>“Catching”</th>
<th>“Throwing”</th>
<th>Non-Interrupting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Escalation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cancel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compensation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Link</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terminate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Activity**

An Activity is a generic term for work that company performs (see page 151) in a Process. An Activity can be atomic or non-atomic (compound). The types of Activities that are a part of a Process Model are: Sub-Process and Task, which are rounded rectangles. Activities are used in both standard Processes and in Choreographies.

**Task (Atomic)**

A Task is an atomic Activity that is included within a Process (see page 156). A Task is used when the work in the Process is not broken down to a finer level of Process detail.

**Choreography Task**

A Choreography Task is an atomic Activity in a Choreography (see page 323). It represents a set of one (1) or more Message exchanges. Each Choreography Task involves two (2) Participants. The name of the Choreography Task and each of the Participants are all displayed in the different bands that make up the shape’s graphical notation. There are two (2) or more Participant Bands and one Task Name Band.
A Sub-Process is a compound Activity that is included within a Process (see page 173) or Choreography (see page 328). It is compound in that it can be broken down into a finer level of detail (a Process or Choreography) through a set of sub-Activities.

**Collapsed Sub-Process**
The details of the Sub-Process are not visible in the Diagram (see page 173). A “plus” sign in the lower-center of the shape indicates that the Activity is a Sub-Process and has a lower-level of detail.

**Expanded Sub-Process**
The boundary of the Sub-Process is expanded and the details (a Process) are visible within its boundary (see page 173). Note that Sequence Flows cannot cross the boundary of a Sub-Process.

**Collapsed Sub-Choreography**
The details of the Sub-Choreography are not visible in the Diagram (see page 328). A “plus” sign in the lower-center of the Task Name Band of the shape indicates that the Activity is a Sub-Process and has a lower-level of detail.

**Expanded Sub-Choreography**
The boundary of the Sub-Choreography is expanded and the details (a Choreography) are visible within its boundary (see page 328). Note that Sequence Flows cannot cross the boundary of a Sub-Choreography.

### Table 7.2 - BPMN Extended Modeling Elements

<table>
<thead>
<tr>
<th>Process/Sub-Process (non-atomic)</th>
<th>See Next Four Figures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collapsed Sub-Process</td>
<td></td>
</tr>
<tr>
<td>Expanded Sub-Process</td>
<td></td>
</tr>
<tr>
<td>Collapsed Sub-Choreography</td>
<td></td>
</tr>
<tr>
<td>Expanded Sub-Choreography</td>
<td></td>
</tr>
</tbody>
</table>
### Table 7.2 - BPMN Extended Modeling Elements

| Gateway | A Gateway is used to control the divergence and convergence of Sequence Flows in a Process (see page 287) and in a Choreography (see page 344). Thus, it will determine branching, forking, merging, and joining of paths. Internal markers will indicate the type of behavior control (see below). |
|---------|
| Gateway Control Types | Icons within the diamond shape of the Gateway will indicate the type of flow control behavior. The types of control include:
- Exclusive decision and merging. Both Exclusive (see page 290) and Event-Based (see page 297) perform exclusive decisions and merging. Exclusive can be shown with or without the “X” marker.
- Event-Based and Parallel Event-based gateways can start a new instance of the Process.
- Inclusive Gateway decision and merging (see page 292).
- Complex Gateway -- complex conditions and situations (e.g., 3 out of 5; page 295).
- Parallel Gateway forking and joining (see page 293).
Each type of control affects both the incoming and outgoing flow. |
| Sequence Flow | A Sequence Flow is used to show the order that Activities will be performed in a Process (see page 97) and in a Choreography (see page 320). |
| Normal Flow | Normal flow refers to paths of Sequence Flow that do not start from an Intermediate Event attached to the boundary of an Activity. |
| Uncontrolled flow | Uncontrolled flow refers to flow that is not affected by any conditions or does not pass through a Gateway. The simplest example of this is a single Sequence Flow connecting two Activities. This can also apply to multiple Sequence Flows that converge to or diverge from an Activity. For each uncontrolled Sequence Flows a token will flow from the source object through the Sequence Flows to the target object. |
### Table 7.2 - BPMN Extended Modeling Elements

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional flow</td>
<td>A Sequence Flow can have a condition expression that are evaluated at runtime to determine whether or not the Sequence Flow will be used (i.e., will a token travel down the Sequence Flow – see page 97). If the conditional flow is outgoing from an Activity, then the Sequence Flow will have a mini-diamond at the beginning of the connector (see figure to the right). If the conditional flow is outgoing from a Gateway, then the line will not have a mini-diamond (see figure in the row above).</td>
</tr>
<tr>
<td>Default flow</td>
<td>For Data-Based Exclusive Gateways or Inclusive Gateways, one type of flow is the Default condition flow (see page 97). This flow will be used only if all the other outgoing conditional flow is not true at runtime. These Sequence Flows will have a diagonal slash will be added to the beginning of the connector (see the figure to the right).</td>
</tr>
<tr>
<td>Exception Flow</td>
<td>Exception flow occurs outside the normal flow of the Process and is based upon an Intermediate Event attached to the boundary of an Activity that occurs during the performance of the Process (see page 287).</td>
</tr>
<tr>
<td>Message Flow</td>
<td>A Message Flow is used to show the flow of Messages between two Participants that are prepared to send and receive them (see page 120). In BPMN, two separate Pools in a Collaboration Diagram will represent the two Participants (e.g., PartnerEntities and/or PartnerRoles).</td>
</tr>
<tr>
<td>Compensation Association</td>
<td>Compensation Association occurs outside the normal flow of the Process and is based upon a Compensation Intermediate Event that is triggered through the failure of a transaction or a throw Compensation Event (see page 302). The target of the Association MUST be marked as a Compensation Activity.</td>
</tr>
</tbody>
</table>
Data Object

Data Objects provide information about what Activities require to be performed and/or what they produce (see page 205). Data Objects can represent a singular object or a collection of objects. Data Input and Data Output provide the same information for Processes.

Message

A Message is used to depict the contents of a communication between two Participants (as defined by a business PartnerRole or a business PartnerEntity—see on page 93).

Fork

BPMN uses the term “fork” to refer to the dividing of a path into two or more parallel paths (also known as an AND-Split). It is a place in the Process where activities can be performed concurrently, rather than sequentially.

There are two options:

- Multiple Outgoing Sequence Flows can be used (see figure top-right). This represents “uncontrolled” flow is the preferred method for most situations.
- A Parallel Gateway can be used (see figure bottom-right). This will be used rarely, usually in combination with other Gateways.

### Table 7.2 - BPMN Extended Modeling Elements

<table>
<thead>
<tr>
<th>Data Object</th>
<th>Data Objec (Collection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Input</td>
<td>Data Output</td>
</tr>
</tbody>
</table>

| Message     | ![Message Icon]         |

| Fork        | ![Fork Diagram]         |

<table>
<thead>
<tr>
<th>![Data Object Icon]</th>
<th>![Data Input Icon]</th>
<th>![Data Output Icon]</th>
</tr>
</thead>
</table>

![Data Object Diagram]

![](BPMN_diagram.png)
**Table 7.2 - BPMN Extended Modeling Elements**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Join</strong></td>
<td>BPMN uses the term “join” to refer to the combining of two or more parallel paths into one path (also known as an AND-Join or synchronization). A Parallel Gateway is used to show the joining of multiple Sequence Flows.</td>
</tr>
<tr>
<td><strong>Decision, Branching Point</strong></td>
<td>Decisions are Gateways within a Process (see page 287) or a Choreography (see page 344) where the flow of control can take one or more alternative paths.</td>
</tr>
<tr>
<td><strong>Exclusive</strong></td>
<td>This Decision represents a branching point where Alternatives are based on conditional Expressions contained within the outgoing Sequence Flows (see page 290 or page 345). Only one of the Alternatives will be chosen.</td>
</tr>
</tbody>
</table>
| **Event-Based**  | This Decision represents a branching point where Alternatives are based on an Event that occurs at that point in the Process (see page 297) or Choreography (see page 350). The specific Event, usually the receipt of a Message, determines which of the paths will be taken. Other types of Events can be used, such as Timer. Only one of the Alternatives will be chosen. There are two options for receiving Messages:  
  - Tasks of Type Receive can be used (see figure top-right).  
  - Intermediate Events of Type Message can be used (see figure bottom-right). |

See next five rows.
Table 7.2 - BPMN Extended Modeling Elements

| Inclusive | This Decision represents a branching point where Alternatives are based on conditional Expressions contained within the outgoing Sequence Flows (see page 292). In some sense it is a grouping of related independent Binary (Yes/No) Decisions. Since each path is independent, all combinations of the paths MAY be taken, from zero to all. However, it should be designed so that at least one path is taken. A Default Condition could be used to ensure that at least one path is taken. There are two versions of this type of Decision:  
  • The first uses a collection of conditional Sequence Flows, marked with mini-diamonds (see top-right figure).  
  • The second uses an Inclusive Gateway (see bottom-right picture). |

| Merging | BPMN uses the term “merge” to refer to the exclusive combining of two or more paths into one path (also known as an OR-Join). A Merging Exclusive Gateway is used to show the merging of multiple Sequence Flows (see upper figure to the right). If all the incoming flow is alternative, then a Gateway is not needed. That is, uncontrolled flow provides the same behavior (see lower figure to the right). |

| Looping | BPMN provides two mechanisms for looping within a Process. See Next Two Figures |

| Activity Looping | The attributes of Tasks and Sub-Processes will determine if they are repeated or performed once (see page 190). There are two types of loops: Standard and Multi-Instance. A small looping indicator will be displayed at the bottom-center of the activity. |
### Table 7.2 - BPMN Extended Modeling Elements

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Flow Looping</td>
<td>Loops can be created by connecting a Sequence Flow to an “upstream” object. An object is considered to be upstream if that object has an outgoing Sequence Flow that leads to a series of other Sequence Flows, the last of which is an incoming Sequence Flow for the original object.</td>
</tr>
<tr>
<td>Multiple Instances</td>
<td>The attributes of Tasks and Sub-Processes will determine if they are repeated or performed once (see page 191). A set of three horizontal lines will be displayed at the bottom-center of the activity for sequential Multi-Instances (see upper figure to the right). A set of three vertical lines will be displayed at the bottom-center of the activity for sequential Multi-Instances (see lower figure to the right).</td>
</tr>
<tr>
<td>Process Break</td>
<td>A Process Break is a location in the Process that shows where an expected delay will occur within a Process (see page 249). An Intermediate Event is used to show the actual behavior (see top-right figure). In addition, a Process Break Artifact, as designed by a modeler or modeling tool, can be associated with the Event to highlight the location of the delay within the flow.</td>
</tr>
<tr>
<td>Transaction</td>
<td>A transaction is a Sub-Process that is supported by a special protocol that insure that all parties involved have complete agreement that the activity should be completed or cancelled (see page 178). The attributes of the activity will determine if the activity is a transaction. A double-lined boundary indicates that the Sub-Process is a Transaction.</td>
</tr>
<tr>
<td>Table 7.2 - BPMN Extended Modeling Elements</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>Nested/Embedded Sub-Process (Inline Block)</strong></td>
<td>A nested (or embedded) Sub-Process is an activity that shares the same set of data as its parent process (see page 173). This is opposed to a Sub-Process that is independent, re-usable, and referenced from the parent process. Data needs to be passed to the referenced Sub-Process, but not to the nested Sub-Process.</td>
</tr>
<tr>
<td><strong>Group (a box around a group of objects within the same category)</strong></td>
<td>A Group is a grouping of graphical elements that are within the same Category (see page 68). This type of grouping does not affect the Sequence Flows within the Group. The Category name appears on the diagram as the group label. Categories can be used for documentation or analysis purposes. Groups are one way in which Categories of objects can be visually displayed on the diagram.</td>
</tr>
<tr>
<td><strong>Off-Page Connector</strong></td>
<td>Generally used for printing, this object will show where a Sequence Flow leaves one page and then restarts on the next page. A Link Intermediate Event can be used as an Off-Page Connector.</td>
</tr>
<tr>
<td><strong>Association</strong></td>
<td>An Association is used to link information and Artifacts with BPMN graphical elements (see page 67). Text Annotations (see page 71) and other Artifacts (see page 66) can be Associated with the graphical elements. An arrowhead on the Association indicates a direction of flow (e.g., data), when appropriate.</td>
</tr>
<tr>
<td><strong>Text Annotation (attached with an Association)</strong></td>
<td>Text Annotations are a mechanism for a modeler to provide additional text information for the reader of a BPMN Diagram (see page 71).</td>
</tr>
<tr>
<td><strong>Pool</strong></td>
<td>A Pool is the graphical representation of a Participant in a Collaboration (see page 112). It also acts as a “swimlane” and a graphical container for partitioning a set of Activities from other Pools, usually in the context of B2B situations. A Pool MAY have internal details, in the form of the Process that will be executed. Or a Pool MAY have no internal details, i.e., it can be a “black box.”</td>
</tr>
</tbody>
</table>
Table 7.2 - BPMN Extended Modeling Elements

| Lanes | A Lane is a sub-partition within a Pool and will extend the entire length of the Pool, either vertically or horizontally (see on page 305). Lanes are used to organize and categorize Activities. |

7.3 BPMN Diagram Types

The BPMN 2.0 aims to cover three basic models of Processes: private Processes (both executable and non-executable), public Processes, and Choreographies. Within and between these three BPMN sub-models, many types of Diagrams can be created. The following are examples of Business Processes that can be modeled using BPMN 2.0:

- High-level non-executable Process Activities (not functional breakdown)
- Detailed executable Business Process
- As-is or old Business Process
- To-be or new Business Process
- A description of expected behavior between two (2) or more business Participants—a Choreography.
- Detailed private Business Process (either executable or non-executable) with interactions to one or more external Entities (or “Black Box” Processes)
- Two or more detailed executable Processes interacting
- Detailed executable Business Process relationship to a Choreography
- Two or more public Processes
- Public Process relationship to Choreography
- Two or more detailed executable Business Processes interacting through a Choreography

BPMN is designed to allow describing all above examples of Business Processes. However, the ways that different sub-models are combined is left to tool vendors. A BPMN 2.0 compliant implementation could RECOMMEND that modelers pick a focused purpose, such as a private Process, or Choreographies. However, the BPMN 2.0 specification makes no assumptions.

7.4 Use of Text, Color, Size, and Lines in a Diagram

Text Annotation objects can be used by the modeler to display additional information about a Process or attributes of the objects within a BPMN Diagram.

- BPMN elements (e.g., Flow objects) MAY have labels (e.g., its name and/or other attributes) placed inside the shape, or above or below the shape, in any direction or location, depending on the preference of the modeler or modeling tool vendor.
- The fills that are used for the graphical elements MAY be white or clear.
- The notation MAY be extended to use other fill colors to suit the purpose of the modeler or tool (e.g., to highlight the value of an object attribute). However,
The markers for “throwing” Events MUST have a dark fill (see “End Event” on page 246 and “Intermediate Event” on page 249 for more details).

Participant Bands for Choreography Tasks and Sub-Choreographies that are not the initiator of the Activity MUST have a light fill (see “Choreography Task” on page 323 and “Sub-Choreography” on page 328 for more details).

Flow objects and markers MAY be of any size that suits the purposes of the modeler or modeling tool.

The lines that are used to draw the graphical elements MAY be black.

The notation MAY be extended to use other line colors to suit the purpose of the modeler or tool (e.g., to highlight the value of an object attribute).

The notation MAY be extended to use other line styles to suit the purpose of the modeler or tool (e.g., to highlight the value of an object attribute) with the condition that the line style MUST NOT conflict with any current BPMN defined line style. Thus, the line styles of Sequence Flows, Message Flows, and Text Associations MUST NOT be modified or duplicated.

7.5 Flow Object Connection Rules

An incoming Sequence Flow can connect to any location on a Flow Object (left, right, top, or bottom). Likewise, an outgoing Sequence Flow can connect from any location on a Flow Object (left, right, top, or bottom). A Message Flow also has this capability. BPMN allows this flexibility; however, we also RECOMMEND that modelers use judgment or best practices in how Flow Objects should be connected so that readers of the Diagrams will find the behavior clear and easy to follow. This is even more important when a Diagram contains Sequence Flows and Message Flows. In these situations it is best to pick a direction of Sequence Flows, either left to right or top to bottom, and then direct the Message Flows at a 90° angle to the Sequence Flows. The resulting Diagrams will be much easier to understand.

7.5.1 Sequence Flow Connections Rules

Table 7.3 displays the BPMN Flow Objects and shows how these objects can connect to one another through Sequence Flows. These rules apply to the connections within a Process Diagram and within a Choreography Diagram. The arrow symbol indicates that the object listed in the row can connect to the object listed in the column. The quantity of connections into and out of an object is subject to various configuration dependencies are not specified here. Refer to the sections in the next chapter for each individual object for more detailed information on the appropriate connection rules. Note that if a Sub-Process has been expanded within a Diagram, the objects within the Sub-Process cannot be connected to objects outside of the Sub-Process. Nor can Sequence Flows cross a Pool boundary.

Table 7.3 – Sequence Flow Connection Rules

<table>
<thead>
<tr>
<th>From\To</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>○</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Only those objects that can have incoming and/or outgoing Sequence Flows are shown in the table. Thus, Pool, Lane, Data Object, Group, and Text Annotation are not listed in the table. Also, the Activity shapes in the table represent Activities and Sub-Processes for Processes, and Choreography Activities and Sub-Choreographies for Choreography.

### 7.5.2 Message Flow Connection Rules

Table 7.4 displays the BPMN modeling objects and shows how these objects can connect to one another through Message Flows. These rules apply to the connections within a Collaboration Diagram. The \( \rightarrow \) symbol indicates that the object listed in the row can connect to the object listed in the column. The quantity of connections into and out of an object is subject to various configuration dependencies that are not specified here. Refer to the sections in the next chapter for each individual object for more detailed information on the appropriate connection rules. *Note that Message Flows cannot connect to objects that are within the same Pool.*
Only those objects that can have incoming and/or outgoing Message Flows are shown in the table. Thus, Lane, Gateway, Data Object, Group, and Text Annotation are not listed in the table.

### 7.6 BPMN Extensibility

**BPMN 2.0** introduces an extensibility mechanism that allows extending standard BPMN elements with additional attributes. It can be used by modelers and modeling tools to add non-standard elements or Artifacts to satisfy a specific need, such as the unique requirements of a vertical domain, and still have valid BPMN Core. Extension attributes MUST NOT contradict the semantics of any BPMN element. In addition, while extensible, BPMN Diagrams should still have the basic look-and-feel so that a Diagram by any modeler should be easily understood by any viewer of the Diagram. Thus the footprint of the basic flow elements (Events, Activities, and Gateways) MUST NOT be altered.

The specification differentiates between mandatory and optional extensions (Section 8.2.3 explains the syntax used to declare extensions). If a mandatory extension is used, a compliant implementation MUST understand the extension. If an optional extension is used, a compliant implementation MAY ignore the extension.
7.7 BPMN Example

The following is an example of a manufacturing process from different perspectives.

Figure 7.6 - An example of a Collaboration diagram with black-box Pools
Figure 7.7 - An example of a stand-alone Choreography diagram
Figure 7.8 - An example of a stand-alone Process (Orchestration) diagram
8 BPMN Core Structure

Note – The content of this chapter is REQUIRED for all BPMN conformance types. For more information about BPMN conformance types, see page 2.

The technical structuring of BPMN is based on the concept of extensibility layers on top of a basic series of simple elements identified as Core Elements of the specification. From this core set of constructs, layering is used to describe additional elements of the specification that extend and add new constructs to the specification and relies on clear dependency paths for resolution. The XML Schema model lends itself particularly well to the structuring model with formalized import and resolution mechanics that remove ambiguities in the definitions of elements in the outer layers of the specification.

Figure 8.1 - A representation of the BPMN Core and Layer Structure

Figure 8.1 shows the basic principles of layering that can be composed in well defined ways. The approach uses formalization constructs for extensibility that are applied consistently to the definition.

The additional effect of layering is that compatibility layers can be built, allowing for different levels of compliance among vendors, and also enabling vendors to add their own layers in support of different vertical industries or target audiences. In addition, it provides mechanism for the redefinition of previously existing concepts without affecting backwards compatibility, but defining two or more non-composable layers, the level of compliance with the specification and backwards compatibility can be achieved without compromising clarity.

The BPMN specification is structured in layers, where each layer builds on top of and extends lower layers. Included is a Core or kernel that includes the most fundamental elements of BPMN, which are REQUIRED for constructing BPMN diagrams: Process, Choreography, and Collaboration. The Core is intended to be simple, concise, and extendable with well defined behavior.
The Core contains three sub-packages (see Figure 8.2):

1. **Foundation**: The fundamental constructs needed for BPMN modeling.
2. **Service**: The fundamental constructs needed for modeling services and interfaces.
3. **Common**: Those classes which are common to the layers of **Process, Choreography, and Collaboration**.

![Class diagram showing the core packages](image)

**Figure 8.2 - Class diagram showing the core packages**

Figure 8.3 displays the organization of the main set of BPMN core model elements.
8.1 Infrastructure

The BPMN Infrastructure package contains two elements that are used for both abstract syntax models and diagram models.

8.1.1 Definitions

The Definitions class is the outermost containing object for all BPMN elements. It defines the scope of visibility and the namespace for all contained elements. The interchange of BPMN files will always be through one or more Definitions.
Figure 8.4 - Definitions class diagram

The Definitions element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 8.1 presents the additional attributes and model associations of the Definitions element.
Table 8.1 - Definitions attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>The name of the Definition.</td>
</tr>
<tr>
<td>targetNamespace: string</td>
<td>This attribute identifies the namespace associated with the Definition and follows the convention established by XML Schema.</td>
</tr>
<tr>
<td>expressionLanguage: string [0..1]</td>
<td>This attribute identifies the formal Expression language used in Expressions within the elements of this Definition. The Default is &quot;<a href="http://www.w3.org/1999/XPath">http://www.w3.org/1999/XPath</a>&quot;. This value MAY be overridden on each individual formal Expression. The language MUST be specified in a URI format.</td>
</tr>
<tr>
<td>typeLanguage: string [0..1]</td>
<td>This attribute identifies the type system used by the elements of this Definition. Defaults to <a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a>. This value can be overridden on each individual ItemDefinition. The language MUST be specified in a URI format.</td>
</tr>
<tr>
<td>rootElements: RootElement [0..*]</td>
<td>This attribute lists the root elements that are at the root of this Definitions. These elements can be referenced within this Definitions and are visible to other Definitions.</td>
</tr>
<tr>
<td>diagrams: BPMNDiagram [0..*]</td>
<td>This attribute lists the BPMNDiagrams that are contained within this Definitions (see page 367 for more information on BPMNDiagrams).</td>
</tr>
<tr>
<td>imports: Import [0..*]</td>
<td>This attribute is used to import externally defined elements and make them available for use by elements within this Definitions.</td>
</tr>
<tr>
<td>extensions: Extension [0..*]</td>
<td>This attribute identifies extensions beyond the attributes and model associations in the base BPMN specification. See page 57 for additional information on extensibility.</td>
</tr>
<tr>
<td>relationships: Relationship [0..*]</td>
<td>This attribute enables the extension and integration of BPMN models into larger system/development Processes.</td>
</tr>
<tr>
<td>exporter: string [0..1]</td>
<td>This attribute identifies the tool that is exporting the bpmn model file.</td>
</tr>
<tr>
<td>exporterVersion: string [0..1]</td>
<td>This attribute identifies the version of the tool that is exporting the bpmn model file.</td>
</tr>
</tbody>
</table>

8.1.2 Import

The Import class is used when referencing external element, either BPMN elements contained in other BPMN Definitions or non-BPMN elements. Imports MUST be explicitly defined.

Table 8.2 presents the attributes of Import.
### Table 8.2 – Import attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>importType: string</td>
<td>Identifies the type of document being imported by providing an absolute URI that identifies the encoding language used in the document. The value of the importType attribute MUST be set to <a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a> when importing XML Schema 1.0 documents, to <a href="http://www.w3.org/TR/wsdl20/">http://www.w3.org/TR/wsdl20/</a> when importing WSDL 2.0 documents, and <a href="http://www.omg.org/spec/BPMN/20100524/MODEL">http://www.omg.org/spec/BPMN/20100524/MODEL</a> when importing BPMN 2.0 documents. Other types of documents MAY be supported. Importing Xml Schema 1.0, WSDL 2.0 and BPMN 2.0 types MUST be supported.</td>
</tr>
<tr>
<td>location: string [0..1]</td>
<td>Identifies the location of the imported element.</td>
</tr>
<tr>
<td>namespace: string</td>
<td>Identifies the namespace of the imported element.</td>
</tr>
</tbody>
</table>

### 8.1.3 Infrastructure Package XML Schemas

### Table 8.3 – Definitions XML schema

```xml
<xsd:element name="definitions" type="tDefinitions"/>
<xsd:complexType name="tDefinitions">
  <xsd:sequence>
    <xsd:element ref="import" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="extension" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="rootElement" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="bpmndi:BPMNDiagram" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="relationship" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="optional"/>
  <xsd:attribute name="targetNamespace" type="xsd:anyURI" use="required"/>
  <xsd:attribute name="expressionLanguage" type="xsd:anyURI" use="optional" default="http://www.w3.org/1999/XPath"/>
  <xsd:attribute name="typeLanguage" type="xsd:anyURI" use="optional" default="http://www.w3.org/2001/XMLSchema"/>
  <xsd:anyAttribute name="exporter" type="xsd:ID"/>
  <xsd:anyAttribute name="exporterVersion" type="xsd:ID"/>
  <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>
```
### Table 8.4 – Import XML schema

```xml
<xsd:element name="import" type="tImport"/>
<xsd:complexType name="tImport">
  <xsd:attribute name="namespace" type="xsd:anyURI" use="required"/>
  <xsd:attribute name="location" type="xsd:string" use="required"/>
  <xsd:attribute name="importType" type="xsd:anyURI" use="required"/>
</xsd:complexType>
```

### 8.2 Foundation

The Foundation package contains classes that are shared among other packages in the Core (see Figure 8.5) of an abstract syntax model.

![Figure 8.5 - Classes in the Foundation package]
8.2.1 Base Element

BaseElement is the abstract super class for most BPMN elements. It provides the attributes id and documentation, which other elements will inherit.

Table 8.5 presents the attributes and model associations for the BaseElement.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>id: string</td>
<td>This attribute is used to uniquely identify BPMN elements. The id is REQUIRED if this element is referenced or intended to be referenced by something else. If the element is not currently referenced and is never intended to be referenced, the id MAY be omitted.</td>
</tr>
<tr>
<td>documentation:</td>
<td>This attribute is used to annotate the BPMN element, such as descriptions and other documentation.</td>
</tr>
<tr>
<td>extensionDefinitions:</td>
<td>This attribute is used to attach additional attributes and associations to any BaseElement. This association is not applicable when the XML schema interchange is used, since the XSD mechanisms for supporting anyAttribute and any element already satisfy this requirement. See page 57 for additional information on extensibility.</td>
</tr>
<tr>
<td>extensionValues:</td>
<td>This attribute is used to provide values for extended attributes and model associations. This association is not applicable when the XML schema interchange is used, since the XSD mechanisms for supporting anyAttribute and any element already satisfy this requirement. See page 57 for additional information on extensibility.</td>
</tr>
</tbody>
</table>

8.2.2 Documentation

All BPMN elements that inherit from the BaseElement will have the capability, through the Documentation element, to have one or more text descriptions of that element.

The Documentation element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 8.6 presents the additional attributes of the Documentation element.

Table 8.6 – Documentation attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>text: string</td>
<td>This attribute is used to capture the text descriptions of a BPMN element.</td>
</tr>
<tr>
<td>textFormat: string</td>
<td>This attribute identifies the format of the text. It MUST follow the mime-type format. The default is &quot;text/plain.&quot;</td>
</tr>
</tbody>
</table>
In the BPMN schema, the Documentation complexType does not contain a text attribute or element. Instead, the documentation text is expected to appear in the body of the documentation element. For example:

<documentation>An example of how the documentation text is entered.</documentation>

### 8.2.3 Extensibility

The BPMN metamodel is aimed to be extensible. This allows BPMN adopters to extend the specified metamodel in a way that allows them to be still BPMN-compliant.

It provides a set of extension elements, which allows BPMN adopters to attach additional attributes and elements to standard and existing BPMN elements.

This approach results in more interchangeable models, because the standard elements are still intact and can still be understood by other BPMN adopters. It’s only the additional attributes and elements that MAY be lost during interchange.

---

**Figure 8.6 - Extension class diagram**

A BPMN Extension basically consists of four different elements:

1. Extension
2. ExtensionDefinition
3. ExtensionAttributeDefinition
4. ExtensionAttributeValue

The core elements of an Extension are the ExtensionDefinition and ExtensionAttributeDefinition. The latter defines a list of attributes that can be attached to any BPMN element. The attribute list defines the name and type of the new attribute. This allows BPMN adopters to integrate any meta model into the BPMN meta model and reuse already existing model elements.

The ExtensionDefinition itself can be created independent of any BPMN element or any BPMN definition.

In order to use an ExtensionDefinition within a BPMN model definition (Definitions element), the ExtensionDefinition MUST be associated with an Extension element that binds the ExtensionDefinition to a specific BPMN model definition. The Extension element itself is contained within the BPMN element Definitions and therefore available to be associated with any BPMN element making use of the ExtensionDefinition.

Every BPMN element which subclasses the BPMN BaseElement can be extended by additional attributes. This works by associating a BPMN element with an ExtensionDefinition, which was defined at the BPMN model definitions level (element Definitions).

Additionally, every “extended” BPMN element contains the actual extension attribute value. The attribute value, defined by the element ExtensionAttributeValue contains the value of type Element. It also has an association to the corresponding attribute definition.

Extension

The Extension element binds/imports an ExtensionDefinition and its attributes to a BPMN model definition.

Table 8.7 presents the attributes and model associations for the Extension element.

Table 8.7 – Extension attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>mustUnderstand</td>
<td>boolean [0..1] = False This flag defines if the semantics defined by the extension definition and its attribute definition MUST be understood by the BPMN adopter in order to process the BPMN model correctly. Defaults to False.</td>
</tr>
<tr>
<td>definition</td>
<td>ExtensionDefinition Defines the content of the extension. Note that in the XML schema, this definition is provided by an external XML schema file and is simply referenced by QName.</td>
</tr>
</tbody>
</table>

ExtensionDefinition

The ExtensionDefinition class defines and groups additional attributes. This type is not applicable when the XML schema interchange is used, since XSD Complex Types already satisfy this requirement.

Table 8.8 presents the attributes and model associations for the ExtensionDefinition element.
ExtensionAttributeDefinition

The ExtensionAttributeDefinition defines new attributes. This type is not applicable when the XML schema interchange is used; since the XSD mechanisms for supporting “AnyAttribute” and “Any” type already satisfy this requirement.

Table 8.9 presents the attributes for the ExtensionAttributeDefinition element.

Table 8.9 - ExtensionAttributeDefinition attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>The name of the extension attribute.</td>
</tr>
<tr>
<td>type: string</td>
<td>The type that is associated with the attribute.</td>
</tr>
<tr>
<td>isReference: boolean</td>
<td>Indicates if the attribute value will be referenced or contained.</td>
</tr>
</tbody>
</table>

ExtensionAttributeValue

The ExtensionAttributeValue contains the attribute value. This type is not applicable when the XML schema interchange is used; since the XSD mechanisms for supporting “AnyAttribute” and “Any” type already satisfy this requirement.

Table 8.10 presents the model associations for the ExtensionAttributeValue element.

Table 8.10 – ExtensionAttributeValue model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: [Element 0..1]</td>
<td>The contained attribute value, used when the associated ExtensionAttributeDefinition.isReference is false. The type of this Element MUST conform to the type specified in the associated ExtensionAttributeDefinition.</td>
</tr>
<tr>
<td>valueRef: [Element 0..1]</td>
<td>The referenced attribute value, used when the associated ExtensionAttributeDefinition.isReference is true. The type of this Element MUST conform to the type specified in the associated ExtensionAttributeDefinition.</td>
</tr>
<tr>
<td>extensionAttributeDefinition: ExtensionAttributeDefinition</td>
<td>Defines the extension attribute for which this value is being provided.</td>
</tr>
</tbody>
</table>
Extensibility XML Schemas

Table 8.11 – Extension XML schema

```xml
<xsd:element name="extension" type="tExtension"/>
<xsd:complexType name="tExtension">
    <xsd:sequence>
        <xsd:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="definition" type="xsd:QName"/>
    <xsd:attribute name="mustUnderstand" type="xsd:boolean" use="optional"/>
</xsd:complexType>
```

XML Example

This example shows a **Task**, defined the **BPMN** Core, being extended with Inputs and Outputs defined outside of the Core.

Table 8.12 – Example Core XML schema

```xml
<xsd:schema ...

    ...
    <xsd:element name="task" type="tTask"/>
    <xsd:complexType name="tTask">
        <xsd:complexContent>
            <xsd:extension base="tActivity"/>
        </xsd:complexContent>
    </xsd:complexType>

    ...
</xsd:schema>
```
Table 8.13 – Example Extension XML schema

```xml
<xsd:schema ...>
...
<xsd:group name="dataRequirements">
  <xsd:sequence>
    <xsd:element ref="dataInput" minOccurs="0" maxOccurs="unbounded" />
    <xsd:element ref="dataOutput" minOccurs="0" maxOccurs="unbounded" />
    <xsd:element ref="inputSet" minOccurs="0" maxOccurs="unbounded" />
    <xsd:element ref="outputSet" minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:group>
...
</xsd:schema>
```

Table 8.14 – Sample XML instance

```xml
<bpmn:definitions id="ID_1" ...>
...
<bpmn:extension mustUnderstand="true" definition="bpmn:dataRequirements"/>
...
<bpmn:task name="Retrieve Customer Record" id="ID_2">
  <bpmn:DataContext name="Order Input" id="ID_3">
    <bpmn:typeDefinition typeRef="bo:Order" id="ID_4"/>
  </bpmn:DataContext>
  <bpmn:DataContext name="Customer Record Output" id="ID_5">
    <bpmn:typeDefinition typeRef="bo:CustomerRecord" id="ID_6"/>
  </bpmn:DataContext>
  <bpmn:inputSet name="Inputs" id="ID_7" dataInputRefs="ID_3"/>
  <bpmn:outputSet name="Outputs" id="ID_8" dataOutputRefs="ID_5"/>
</bpmn:task>
...
</bpmn:definitions>
```

8.2.4 External Relationships

It is the intention of this specification to cover the basic elements necessary for the construction of semantically rich and syntactically valid Process models to be used in the description of Processes, Choreographies, and business operations in multiple levels of abstraction. As the specification indicates, extension capabilities enable the enrichment of the information described in BPMN and supporting models to be augmented to fulfill particularities of a given usage model. These extensions intention is to extend the semantics of a given BPMN Artifact to provide specialization of intent or meaning.
Process models do not exist in isolation and generally participate in larger, more complex business and system development Processes. The intention of the following specification element is to enable BPMN Artifacts to be integrated in these development Processes via the specification of a non-intrusive identity/relationship model between BPMN Artifacts and elements expressed in any other addressable domain model.

The ‘identity/relationship’ model is reduced to the creation of families of typed relationships that enable BPMN and non-BPMN Artifacts to be related in non-intrusive manner. By simply defining ‘relationship types’ that can be associated with elements in the BPMN Artifacts and arbitrary elements in a given addressable domain model, it enables the extension and integration of BPMN models into larger system/development Processes.

It is that these extensions will enable, for example, the linkage of ‘derivation’ or ‘definition’ relationships between UML artifacts and BPMN Artifacts in novel ways. So, a UML use case could be related to a Process element in the BPMN specification without affecting the nature of the Artifacts themselves, but enabling different integration models that traverse specialized relationships.

Simply, the model enables the external specification of augmentation relationships between BPMN Artifacts and arbitrary relationship classification models, these external models, via traversing relationships declared in the external definition allow for linkages between BPMN elements and other structured or non-structured metadata definitions.

The UML model for this specification follow a simple extensible pattern as shown below; where named relationships can be established by referencing objects that exist in their given namespaces.

![Figure 8.7 - External Relationship Metamodel](image)

The Relationship element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 8.15 presents the additional attributes for the Relationship element.
In this manner, you can, for example, create relationships between different artifacts that enable external annotations used for (for example) traceability, derivation, arbitrary classifications, etc.

An example where the ‘reengineer’ relationship is shown between elements in a Visio ™ artifact and a BPMN Artifact.

Table 8.15 – Relationship attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>type: string</td>
<td>The descriptive name of the element.</td>
</tr>
<tr>
<td>direction: RelationshipDirection {None</td>
<td>Forward</td>
</tr>
<tr>
<td>sources: [Element [1..*]]</td>
<td>This association defines artifacts that are augmented by the relationship.</td>
</tr>
<tr>
<td>targets: [Element[1..*]]</td>
<td>This association defines artifacts used to extend the semantics of the source element(s).</td>
</tr>
</tbody>
</table>

An example where the ‘reengineer’ relationship is shown between elements in a Visio ™ artifact and a BPMN Artifact.

Table 8.16 – Reengineer XML schema

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="" typeLanguage="" expressionLanguage=""
 xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL Core-Common.xsd"
xmns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmns:src="http://www.example.org/Processes/Old"
xmns:tgx="http://www.example.org/Processes/New">

<import importType="http://office.microsoft.com/visio" location="OrderConfirmationProcess.vsd"
 namespace="http://www.example.org/Processes/Old"/>

<import importType="http://www.omg.org/spec/BPMN/20100524/MODEL"
 location="OrderConfirmationProcess.xml"
 namespace="http://www.example.org/Processes/New"/>

<relationship type="reengineered" id="a234" direction="both">
 <documentation>An as-is and to-be relationship. The as-is model is expressed as a Visio diagram. The re-engineered process has been split in two and is captured in BPMN 2.0 format.</documentation>
 <source ref="src:OrderConfirmation"/>
 <target ref="tgt:OrderConfirmation_PartI"/>
 <target ref="tgt:OrderConfirmation_PartII"/>
</relationship>
</definitions>
8.2.5 Root Element

RootElement is the abstract super class for all BPMN elements that are contained within Definitions. When contained within Definitions, these elements have their own defined life-cycle and are not deleted with the deletion of other elements. Examples of concrete RootElements include Collaboration, Process, and Choreography. Depending on their use, RootElements can be referenced by multiple other elements (i.e., they can be reused). Some RootElements MAY be contained within other elements instead of Definitions. This is done to avoid the maintenance overhead of an independent life-cycle. For example, an EventDefinition would be contained in a Process since it is used only there. In this case the EventDefinition would be dependent on the tool life-cycle of the Process.

The RootElement element inherits the attributes and model associations of BaseElement (see Table 8.5), but does not have any further attributes or model associations.

8.2.6 Foundation Package XML Schemas

Table 8.17 – BaseElement XML schema

```xml
<xsd:element name="baseElement" type="tBaseElement"/>
<xsd:complexType name="tBaseElement" abstract="true">
  <xsd:sequence>
    <xsd:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="extensionElements" minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="optional"/>
  <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>

<xsd:element name="baseElementWithMixedContent" type="tBaseElementWithMixedContent"/>
<xsd:complexType name="tBaseElementWithMixedContent" abstract="true" mixed="true">
  <xsd:sequence>
    <xsd:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="extensionElements" minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="optional"/>
  <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>

<xsd:element name="extensionElements" type="tExtensionElements"/>
<xsd:complexType name="tExtensionElements">
  <xsd:sequence>
    <xsd:any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="documentation" type="tDocumentation"/>
```
Table 8.18 – RootElement XML schema

```xml
<xsd:element name="rootElement" type="tRootElement"/>
<xsd:complexType name="tRootElement" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8.19 – Relationship XML schema

```xml
<xsd:element name="relationship" type="tRelationship"/>
<xsd:complexType name="tRelationship">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="source" type="xsd:QName" minOccurs="1" maxOccurs="unbounded"/>
                <xsd:element name="target" type="xsd:QName" minOccurs="1" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="type" type="xsd:string" use="required"/>
            <xsd:attribute name="direction" type="tRelationshipDirection"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

```xml
<xsd:simpleType name="tRelationshipDirection">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="None"/>
        <xsd:enumeration value="Forward"/>
        <xsd:enumeration value="Backward"/>
        <xsd:enumeration value="Both"/>
    </xsd:restriction>
</xsd:simpleType>
```
8.3 Common Elements

The following sections define BPMN elements that MAY be used in more than one type of diagram (e.g., Process, Collaboration, and Choreography).

8.3.1 Artifacts

BPMN provides modelers with the capability of showing additional information about a Process that is not directly related to the Sequence Flows or Message Flows of the Process.

At this point, BPMN provides three standard Artifacts: Associations, Groups, and Text Annotations. Additional Artifacts MAY be added to the BPMN specification in later versions. A modeler or modeling tool MAY extend a BPMN diagram and add new types of Artifacts to a Diagram. Any new Artifact MUST follow the Sequence Flow and Message Flow connection rules (listed below). Associations can be used to link Artifacts to Flow Objects (see page 67).

Figure 8.8 shows the Artifacts class diagram. When an Artifact is defined it is contained within a Collaboration or a FlowElementsContainer (a Process or Choreography).

![Artifacts Metamodel](image)

Common Artifact Definitions

The following sections provide definitions that are common to all Artifacts.

Artifact Sequence Flow Connections

See “Sequence Flow Connections Rules” on page 42 for the entire set of objects and how they MAY be source or targets of a Sequence Flow.
• An Artifact MUST NOT be a target for a **Sequence Flow**.
• An Artifact MUST NOT be a source for a **Sequence Flow**.

**Artifact Message Flow Connections**
See “Message Flow Connection Rules” on page 43 for the entire set of objects and how they MAY be source or targets of a **Message Flow**.
• An Artifact MUST NOT be a target for a **Message Flow**.
• An Artifact MUST NOT be a source for a **Message Flow**.

**Association**
An **Association** is used to associate information and **Artifacts** with **Flow Objects**. Text and graphical non-Flow Objects can be associated with the Flow Objects and Flow. An **Association** is also used to show the **Activity** used for compensation. More information about compensation can be found on page 302.
• An **Association** is line that MUST be drawn with a dotted single line (see Figure 8.9).
• The use of text, color, size, and lines for an Association MUST follow the rules defined in “Use of Text, Color, Size, and Lines in a Diagram” on page 41.

**Figure 8.9 - An Association**

If there is a reason to put directionality on the **Association** then:
• A line arrowhead MAY be added to the **Association** line (see Figure 8.11).
• The directionality of the **Association** can be in one (1) direction or in both directions.
Figure 8.11 - A Directional Association

Note that directional Associations were used in BPMN 1.2 to show how Data Objects were inputs or outputs to Activities. In BPMN 2.0, a Data Association connector is used to show inputs and outputs (see page 221). A Data Association uses the same notation as a directed Association (as in Figure 8.11, above).

An Association is used to connect user-defined text (an Annotation) with a Flow Object (see Figure 8.12).

Figure 8.12 - An Association of Text Annotation

The Association element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 8.20 presents the additional attributes and model associations for an Association.

Table 8.20 – Association attributes and model associations

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>associationDirection: AssociationDirection = None (None</td>
<td>One</td>
</tr>
<tr>
<td>sourceRef: BaseElement</td>
<td>The BaseElement that the Association is connecting from.</td>
</tr>
<tr>
<td>targetRef: BaseElement</td>
<td>The BaseElement that the Association is connecting to.</td>
</tr>
</tbody>
</table>

Group

The Group object is an Artifact that provides a visual mechanism to group elements of a diagram informally. The grouping is tied to the CategoryValue supporting element. That is, a Group is a visual depiction of a single CategoryValue. The graphical elements within the Group will be assigned the CategoryValue of the Group. (Note -- CategoryValues can be highlighted through other mechanisms, such as color, as defined by a modeler or a modeling tool).

- A Group is a rounded corner rectangle that MUST be drawn with a solid dashed line (as seen in Figure 8.13).
The use of text, color, size, and lines for a **Group** MUST follow the rules defined in “Use of Text, Color, Size, and Lines in a Diagram” on page 41.

---

**Figure 8.13 - A Group Artifact**

As an **Artifact**, a **Group** is not an **Activity** or any **Flow Object**, and, therefore, cannot connect to **Sequence Flows** or **Message Flows**. In addition, **Groups** are not constrained by restrictions of **Pools** and **Lanes**. This means that a **Group** can stretch across the boundaries of a **Pool** to surround **Diagram** elements (see Figure 8.14), often to identify **Activities** that exist within a distributed business-to-business transaction.

---

**Figure 8.14 - A Group around Activities in different Pools**

**Groups** are often used to highlight certain sections of a Diagram without adding additional constraints for performance—**as a Sub-Process** would. The highlighted (grouped) section of the Diagram can be separated for reporting and analysis purposes. **Groups** do not affect the flow of the **Process**.

Figure 8.15 shows the **Group** class diagram.
The Group element inherits the attributes and model associations of BaseElement (see Table 8.5), through its relationship to Artifact. Table 8.21 presents the additional model associations for a Group.

Table 8.21 – Group model associations

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>categoryValueRef: Category-Value</td>
<td>The categoryValueRef attribute specifies the CategoryValue that the Group represents. (Further details about the definition of a Category and CategoryValue can be found on page 70.) The name of the Category and the value of the CategoryValue separated by delineator &quot;.&quot; provides the label for the Group. The graphical elements within the boundaries of the Group will be assigned the CategoryValue.</td>
</tr>
</tbody>
</table>

**Category**

Categories, which have user-defined semantics, can be used for documentation or analysis purposes. For example, FlowElements can be categorized as being customer oriented vs. support oriented. Furthermore, the cost and time of Activities per Category can be calculated.

**Groups** are one way in which Categories of objects can be visually displayed on the diagram. That is, a Group is a visual depiction of a single CategoryValue. The graphical elements within the Group will be assigned the CategoryValue of the Group. The value of the CategoryValue, optionally prepended by the Category name and delineator ".", appears on the diagram as the Group label. (Note -- Categories can be highlighted through other mechanisms, such as color, as defined by a modeler or a modeling tool). A single Category can be used for multiple Groups in a diagram.

The Category element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to RootElement. Table 8.22 displays the additional model associations of the Category element.
The `CategoryValue` element inherits the attributes and model associations of `BaseElement` (see Table 8.5). Table 8.23 displays the attributes and model associations of the `CategoryValue` element.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>The descriptive name of the element.</td>
</tr>
<tr>
<td>categoryValue: CategoryValue [0..*]</td>
<td>The <code>categoryValue</code> attribute specifies one or more values of the <code>Category</code>. For example, the <code>Category</code> is “Region” then this <code>Category</code> could specify values like “North,” “South,” “West,” and “East.”</td>
</tr>
</tbody>
</table>

Table 8.23 – CategoryValue attributes and model associations

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: string</td>
<td>This attribute provides the value of the <code>CategoryValue</code> element.</td>
</tr>
<tr>
<td>category: Category [0..1]</td>
<td>The <code>category</code> attribute specifies the <code>Category</code> representing the <code>Category</code> as such and contains the <code>CategoryValue</code> (Further details about the definition of a <code>Category</code> can be found on page 70).</td>
</tr>
<tr>
<td>categorizedFlowElements: FlowElement [0..*]</td>
<td>The <code>FlowElements attribute identifies all of the elements (e.g., Events, Activities, Gateways, and Artifacts) that are within the boundaries of the </code>Group`.</td>
</tr>
</tbody>
</table>

Text Annotation

Text Annotations are a mechanism for a modeler to provide additional information for the reader of a BPMN Diagram.

- A Text Annotation is an open rectangle that MUST be drawn with a solid single line (as seen in Figure 8.16).
- The use of text, color, size, and lines for a Text Annotation MUST follow the rules defined in “Use of Text, Color, Size, and Lines in a Diagram” on page 41.

The Text Annotation object can be connected to a specific object on the Diagram with an Association, but does not affect the flow of the Process. Text associated with the Annotation can be placed within the bounds of the open rectangle.

Figure 8.16 - A Text Annotation

The Text Annotation element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 8.24 presents the additional attributes for a Text Annotation.
Table 8.24 – Text Annotation attributes

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text: string</td>
<td>Text is an attribute that is text that the modeler wishes to communicate to the reader of the Diagram.</td>
</tr>
<tr>
<td>textFormat: string</td>
<td>This attribute identifies the format of the text. It MUST follow the mime-type format. The default is &quot;text/plain.&quot;</td>
</tr>
</tbody>
</table>

XML Schema for Artifacts

Table 8.25 – Artifact XML schema

```xml
<xsd:element name="artifact" type="tArtifact"/>
<xsd:complexType name="tArtifact" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8.26 – Association XML schema

```xml
<xsd:element name="association" type="tAssociation" substitutionGroup="artifact"/>
<xsd:complexType name="tAssociation">
    <xsd:complexContent>
        <xsd:extension base="tArtifact">
            <xsd:attribute name="sourceRef" type="xsd:QName" use="required"/>
            <xsd:attribute name="targetRef" type="xsd:QName" use="required"/>
            <xsd:attribute name="associationDirection" type="tAssociationDirection" default="None"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

```xml
<xsd:simpleType name="tAssociationDirection">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="None"/>
        <xsd:enumeration value="One"/>
        <xsd:enumeration value="Both"/>
    </xsd:restriction>
</xsd:simpleType>
```

Table 8.27 – Category XML schema

```xml
<xsd:element name="category" type="tCategory" substitutionGroup="RootElement"/>
<xsd:complexType name="tCategory"/>
```
Table 8.28 – CategoryValue XML schema

```xml
<xsd:element name="categoryValue" type="tCategoryValue"/>
<xsd:complexType name="tCategoryValue">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:attribute name="value" type="xsd:string" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 8.29 – Group XML schema

```xml
<xsd:element name="group" type="tGroup" substitutionGroup="artifact"/>
<xsd:complexType name="tGroup">
  <xsd:complexContent>
    <xsd:extension base="tArtifact">
      <xsd:attribute name="categoryValueRef" type="xsd:QName" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 8.30 – Text Annotation XML schema

```xml
<xsd:element name="textAnnotation" type="tTextAnnotation" substitutionGroup="artifact"/>
<xsd:complexType name="tTextAnnotation">
  <xsd:complexContent>
    <xsd:extension base="tArtifact">
      <xsd:sequence>
        <xsd:element ref="text" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="textFormat" type="xsd:string" default="textplain"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
### 8.3.2 Correlation

**Business Processes** typically can run for days or even months, requiring asynchronous communication via **Message**. Also, many **instances** of a particular **Process** will typically run in parallel, e.g., many **instances** of an order process, each representing a particular order. **Correlation** is used to associate a particular **Message** to an ongoing **Conversation** between two particular **Process** instances. **BPMN** allows using existing **Message** data for **correlation** purposes, e.g., for the order process, a particular **instance** can be identified by means of its **orderID** and/or **customerID**, rather than requiring the introduction of technical **correlation** data.

The concept of **Correlation** facilitates the association of a **Message** to a **Send Task** or **Receive Task**\(^1\) often in the context of a **Conversation**, which is also known as **instance routing**. It is a particular useful concept where there is no infrastructure support for **instance** routing. Note that this association can be viewed at multiple levels, namely the **Collaboration** (**Conversation**), **Choreography**, and **Process** level. However, the actual correlation happens during runtime (e.g., at the **Process** level). **Correlations** describe a set of predicates on a **Message** (generally on the application payload) that need to be satisfied in order for that **Message** to be associated to a distinct **Send Task** or **Receive Task**. By the same token, each **Send Task** and each **Receive Task** participates in one or many **Conversations**. Furthermore, it identifies the **Message** it sends or receives and thereby establishes the relationship to one (or many) **CorrelationKeys**.

There are two, non-exclusive correlation mechanisms in place:

1. In plain, key-based correlation, **Messages** that are exchanged within a **Conversation** are logically correlated by means of one or more common **CorrelationKeys**. That is, any **Message** that is sent or received within this **Conversation** needs to carry the value of at least one of these **CorrelationKey** instances within its payload. A **CorrelationKey** basically defines a (composite) key. The first **Message** that is initially sent or received initializes one or more **CorrelationKey** instances associated with the **Conversation**, i.e., assigns values to its **CorrelationProperty** instances which are the fields (partial keys) of the **CorrelationKey**. A **CorrelationKey** is only considered valid for use, if the **Message** has resulted in all **CorrelationProperty** fields within the key being populated with a value. If a follow-up **Message** derives a **CorrelationKey** instance, where that **CorrelationKey** had previously been initialized within the **Conversation**, then the **CorrelationKey** value in the **Message** and **Conversation** MUST match. If the follow-up **Message** derives a **CorrelationKey** instance associated with the **Conversation**, that had not previously been initialized, then the **CorrelationKey** value will become associated with the **Conversation**. As a **Conversation** can comprise different **Messages** that can be differently structured, each **CorrelationProperty** comes with as many extraction rules (**CorrelationPropertyRetrievalExpression**) for the respective partial key as there are different **Messages**.

2. In context-based correlation, the **Process** context (i.e., its **Data Objects** and **Properties**) can dynamically influence the matching criterion. That is, a **CorrelationKey** can be complemented by a **Process**-specific

---

\(^1\) All references to **Send** or **Receive Tasks** in this section also include message **catch** or **throw** **Events** -- they behave identically with respect to correlation.
CorrelationSubscription. A CorrelationSubscription aggregates as many CorrelationPropertyBindings as there are CorrelationProperties in the CorrelationKey. A CorrelationPropertyBinding relates to a specific CorrelationProperty and also links to a FormalExpression that denotes a dynamic extraction rule atop the Process context. At runtime, the CorrelationKey instance for a particular Conversation is populated (and dynamically updated) from the Process context using these FormalExpressions. In that sense, changes in the Process context can alter the correlation condition.

Correlation can be applied to Message Flows in Collaboration and Choreography, as described in Chapters 9, Collaboration and 11, Choreography. The keys applying to a Message Flow are the keys of containers or groupings of the Message Flow, such as Collaborations, Choreographies, and Conversation Nodes, and Choreography Activities. This might result in multiple CorrelationKeys applying to the same Message Flow, perhaps due to multiple layers of containment. In particular, calls of Collaborations and Choreographies are special kinds of Conversation Nodes and Choreography Activities, respectively, and are considered a kind of containment for the purposes of correlation. The CorrelationKeys specified in the caller apply to Message Flow in a called Collaboration or Choreography.
CorrelationKey

A CorrelationKey represents a composite key out of one or many CorrelationProperties that essentially specify extraction Expressions atop Messages. As a result, each CorrelationProperty acts as a partial key for the correlation. For each Message that is exchanged as part of a particular Conversation, the CorrelationProperties need to provide a CorrelationPropertyRetrievalExpression which references a FormalExpression to the Message payload. That is, for each Message (that is used in a Conversation) there is an Expression, which extracts portions of the respective Message's payload.

The CorrelationKey element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 8.31 displays the additional model associations of the CorrelationKey element.
Key-based Correlation

Key-based correlation is a simple and efficient form of correlation, where one or more keys are used to identify a Conversation. Any incoming Message can be matched against the CorrelationKey by extracting the CorrelationProperties from the Message according to the corresponding CorrelationPropertyRetrievalExpression and comparing the resulting composite key with the CorrelationKey instance for this Conversation. The idea is to use a joint Conversation “token” that is used (passed to and received from) and outgoing and incoming Messages are associated to a particular Conversation if the composite key extracted from their payload matches the CorrelationKey initialized for this Conversation.

At runtime the first Send Task or Receive Task in a Conversation MUST populate at least one of the CorrelationKey instances by extracting the values of the CorrelationProperties according to the CorrelationPropertyRetrievalExpression from the initially sent or received Message. Later in the Conversation, the populated CorrelationKey instances are used for the described matching procedure where from incoming Messages a composite key is extracted and used to identify the associated Conversation. Where these non-initiating Messages derive values for CorrelationKeys, associated with the Conversation but not yet populated, then the derived value will be associated with the Conversation instance.

The CorrelationProperty element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to RootElement. Table 8.32 displays the additional model associations of the CorrelationProperty element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string [0..1]</td>
<td>Specifies the name of the CorrelationProperty.</td>
</tr>
<tr>
<td>type: string [0..1]</td>
<td>Specifies the type of the CorrelationProperty.</td>
</tr>
<tr>
<td>correlationPropertyRetrievalExpression: CorrelationPropertyRetrievalExpression [1..*]</td>
<td>The CorrelationPropertyRetrievalExpressions for this CorrelationProperty, representing the associations of FormalExpressions (extraction paths) to specific Messages occurring in this Conversation.</td>
</tr>
</tbody>
</table>

The CorrelationPropertyRetrievalExpression element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 8.33 displays the additional model associations of the CorrelationPropertyRetrievalExpression element.

Table 8.31 – CorrelationKey model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string [0..1]</td>
<td>Specifies the name of the CorrelationKey.</td>
</tr>
<tr>
<td>correlationPropertyRef: CorrelationProperty [0..*]</td>
<td>The CorrelationProperties, representing the partial keys of this CorrelationKey.</td>
</tr>
</tbody>
</table>
Context-based Correlation

Context-based correlation is a more expressive form of correlation on top of key-based correlation. In addition to implicitly populating the CorrelationKey instance from the first sent or received Message, another mechanism relates the CorrelationKey to the Process context. That is, a Process MAY provide a CorrelationSubscription that acts as the Process-specific counterpart to a specific CorrelationKey. In this way, a Conversation MAY additionally refer to explicitly updateable Process context data to determine whether or not a Message needs to be received. At runtime, the CorrelationKey instance holds a composite key that is dynamically calculated from the Process context and automatically updated whenever the underlying Data Objects or Properties change.

CorrelationPropertyBindings represent the partial keys of a CorrelationSubscription where each relates to a specific CorrelationProperty in the associated CorrelationKey. A FormalExpression defines how that CorrelationProperty instance is populated and updated at runtime from the Process context (i.e., its Data Objects and Properties).

The CorrelationSubscription element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 8.34 displays the additional model associations of the CorrelationSubscription element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlationKeyRef: CorrelationKey</td>
<td>The CorrelationKey this CorrelationSubscription refers to.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlationPropertyBinding: CorrelationPropertyBinding [0..*]</td>
<td>The bindings to specific CorrelationProperties and FormalExpressions (extraction rules atop the Process context).</td>
</tr>
</tbody>
</table>

The CorrelationPropertyBinding element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 8.35 displays the additional model associations of the CorrelationPropertyBinding element.
At runtime, the correlation mechanism works as follows: When a Process instance is created the CorrelationKey instances of all Conversations are initialized with some initial values that specify to correlate any incoming Message for these Conversations. A SubscriptionProperty is updated whenever any of the Data Objects or Properties changes that are referenced from the respective FormalExpression. As a result, incoming Messages are matched against the now populated CorrelationKey instance. Later in the Process run, the SubscriptionProperties can, again, change and implicitly change the correlation criterion. Alternatively, the established mechanism of having the first Send Task or Receive Task populate the CorrelationKey instance applies.

**XML Schema for Correlation**

**Table 8.36 – Correlation Key XML schema**

```xml
<xsd:element name="correlationKey" type="tCorrelationKey"/>
<xsd:complexType name="tCorrelationKey">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:sequence>
        <xsd:element name="correlationPropertyRef" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:String" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

**Table 8.37 – Correlation Property XML schema**

```xml
<xsd:element name="correlationProperty" type="tCorrelationProperty" substitutionGroup="rootElement"/>
<xsd:complexType name="tCorrelationProperty">
  <xsd:complexContent base="tRootElement">
    <xsd:sequence>
      <xsd:element ref="correlationPropertyRetrievalExpression" minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:String" use="optional"/>
    <xsd:attribute name="type" type="xsd:QName"/>
  </xsd:complexContent>
</xsd:complexType>
```

**Table 8.35 – CorrelationPropertyBinding model associations**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataPath: FormalExpression</td>
<td>The FormalExpression that defines the extraction rule atop the Process context.</td>
</tr>
<tr>
<td>correlationPropertyRef: CorrelationProperty</td>
<td>The specific CorrelationProperty, this CorrelationPropertyBinding refers to.</td>
</tr>
</tbody>
</table>
### Table 8.38 – Correlation Property Binding XML schema

```xml
<xsd:element name="correlationPropertyBinding" type="tCorrelationPropertyBinding"/>
<xsd:complexType name="tCorrelationPropertyBinding">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:sequence>
        <xsd:element name="dataPath" type="tFormalExpression" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="correlationPropertyRef" type="xsd:QName" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

### Table 8.39 – Correlation Property Retrieval Expression XML schema

```xml
<xsd:element name="correlationPropertyRetrievalExpression" type="tCorrelationPropertyRetrievalExpression"/>
<xsd:complexType name="tCorrelationPropertyRetrievalExpression">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:sequence>
        <xsd:element name="messagePath" type="tFormalExpression" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="messageRef" type="xsd:QName" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

### Table 8.40 – Correlation Subscription XML schema

```xml
<xsd:element name="correlationSubscription" type="tCorrelationSubscription"/>
<xsd:complexType name="tCorrelationSubscription">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:sequence>
        <xsd:element name="process" type="xsd:QName" use="required"/>
        <xsd:element ref="correlationKeyRef" minOccurs="1" maxOccurs="1"/>
        <xsd:element name="correlationPropertyBinding" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
8.3.3 Error

An Error represents the content of an Error Event or the Fault of a failed Operation. An ItemDefinition is used to specify the structure of the Error. An Error is generated when there is a critical problem in the processing of an Activity or when the execution of an Operation failed.

Figure 8.18 - Error class diagram

The Error element inherits the attributes and model associations of BaseElement (see Table 8.5), through its relationship toRootElement. Table 8.41 presents the additional attributes and model associations of the Error element.
8.3.4 Escalation

An Escalation identifies a business situation that a Process might need to react to. An ItemDefinition is used to specify the structure of the Escalation.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>structureRef : ItemDefinition [0..1]</td>
<td>An ItemDefinition is used to define the “payload” of the Error.</td>
</tr>
<tr>
<td>name : string</td>
<td>The descriptive name of the Error.</td>
</tr>
</tbody>
</table>
| errorCode : string | For an End Event:
  - If the result is an Error, then the errorCode MUST be supplied (if the processType attribute of the Process is set to executable) This “throws” the Error.

For an Intermediate Event within normal flow:
  - If the trigger is an Error, then the errorCode MUST be entered (if the processType attribute of the Process is set to executable). This “throws” the Error.

For an Intermediate Event attached to the boundary of an Activity:
  - If the trigger is an Error, then the errorCode MAY be entered. This Event “catches” the Error. If there is no errorCode, then any error SHALL trigger the Event. If there is an errorCode, then only an Error that matches the errorCode SHALL trigger the Event.

Figure 8.19 - Escalation class diagram
The Escalation element inherits the attributes and model associations of BaseElement (see Table 8.5), through its relationship to RootElement. Table 8.41 presents the additional model associations of the Error element.

**Table 8.42 – Escalation attributes and model associations**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>structureRef : ItemDefinition [0..1]</td>
<td>An ItemDefinition is used to define the “payload” of the Escalation.</td>
</tr>
<tr>
<td>name : string</td>
<td>The descriptive name of the Escalation.</td>
</tr>
<tr>
<td>escalationCode: string</td>
<td></td>
</tr>
</tbody>
</table>

For an **End Event**:

If the Result is an Escalation, then the escalationCode MUST be supplied (if the processType attribute of the Process is set to executable). This “throws” the Escalation.

For an **Intermediate Event within normal flow**:

If the trigger is an Escalation, then the escalationCode MUST be entered (if the processType attribute of the Process is set to executable). This “throws” the Escalation.

For an **Intermediate Event attached to the boundary of an Activity**:

If the trigger is an Escalation, then the escalationCode MAY be entered. This Event “catches” the Escalation. If there is no escalationCode, then any Escalation SHALL trigger the Event. If there is an escalationCode, then only an Escalation that matches the escalationCode SHALL trigger the Event.

**8.3.5 Events**

An **Event** is something that “happens” during the course of a **Process**. These **Events** affect the flow of the **Process** and usually have a cause or an impact. The term “event” is general enough to cover many things in a **Process**. The start of an **Activity**, the end of an **Activity**, the change of state of a document, a **Message** that arrives, etc., all could be considered **Events**. However, **BPMN** has restricted the use of **Events** to include only those types of **Events** that will affect the sequence or timing of **Activities** of a **Process**.
The Event element inherits the attributes and model associations of FlowElement (see Table 8.44), but adds no additional attributes or model associations.

The details for the types of Events (Start, Intermediate, and End) are defined in “Event Definitions” on page 260.

8.3.6 Expressions

The Expression class is used to specify an Expression using natural-language text. These Expressions are not executable. The natural language text is captured using the documentation attribute, inherited from BaseElement.

Expression inherits the attributes and model associations of BaseElement (see Table 8.5), but adds no additional attributes or model associations.
Expressions are used in many places within BPMN to extract information from the different elements, normally data elements. The most common usage is when modeling decisions, where conditional Expressions are used to direct the flow along specific paths based on some criteria.

BPMN supports underspecified Expressions, where the logic is captured as natural-language descriptive text. It also supports formal Expressions, where the logic is captured in an executable form using a specified Expression language.

![Expression class diagram](image)

**Figure 8.21 - Expression class diagram**

**Expression**

The Expression class is used to specify an Expression using natural-language text. These Expressions are not executable and are considered underspecified.

The definition of an Expression can be done in two ways: it can be contained where it is used, or it can be defined at the Process level and then referenced where it is used.

The Expression element inherits the attributes and model associations of BaseElement (see Table 8.5), but does not have any additional attributes or model associations.

**Formal Expression**

The FormalExpression class is used to specify an executable Expression using a specified Expression language. A natural-language description of the Expression can also be specified, in addition to the formal specification.

The default Expression language for all Expressions is specified in the Definitions element, using the expressionLanguage attribute. It can also be overridden on each individual FormalExpression using the same attribute.

The FormalExpression element inherits the attributes and model associations of BaseElement (see Table 8.5), through the Expression element. Table 8.43 presents the additional attributes and model associations of the FormalExpression.
8.3.7 Flow Element

FlowElement is the abstract super class for all elements that can appear in a Process flow, which are FlowNodes (see page 99), which consist of Activities (see page 151), Choreography Activities (see page 321), Gateways (see page 287), and Events (see page 233), Data Objects (see page 205), Data Associations (see page 221), and Sequence Flows (see page 97).
Figure 8.22 - FlowElement class diagram

The `FlowElement` element inherits the attributes and model associations of `BaseElement` (see Table 8.5). Table 8.44 presents the additional attributes and model associations of the `FlowElement` element.
### 8.3.8 Flow Elements Container

FlowElementsContainer is an abstract super class for BPMN diagrams (or views) and defines the superset of elements that are contained in those diagrams. Basically, a FlowElementsContainer contains FlowElements, which are **Events** (see page 233), **Gateways** (see page 287), **Sequence Flows** (see page 97), **Activities** (see page 151), and **Choreography Activities** (see page 321).

There are four (4) types of FlowElementsContainers (see Figure 8.23): **Process**, **Sub-Process**, **Choreography**, and **Sub-Choreography**.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string [0..1]</td>
<td>The descriptive name of the element.</td>
</tr>
<tr>
<td>categoryValueRef: Category-Value [0..*]</td>
<td>A reference to the Category Values that are associated with this Flow Element.</td>
</tr>
<tr>
<td>auditing: Auditing [0..1]</td>
<td>A hook for specifying audit related properties. Auditing can only be defined for a Process.</td>
</tr>
<tr>
<td>monitoring: Monitoring [0..1]</td>
<td>A hook for specifying monitoring related properties. Monitoring can only be defined for a Process.</td>
</tr>
</tbody>
</table>
Figure 8.23 - FlowElementContainers class diagram

The FlowElementContainers element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 8.45 presents the additional model associations of the FlowElementContainers element.

Table 8.45 – FlowElementContainers model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
</table>
| flowElements: Flow Element [0..*] | This association specifies the particular flow elements contained in a FlowElementContainer. Flow elements are Events, Gateways, Sequence Flows, Activities, Data Objects, Data Associations, and Choreography Activities. Note that:  
  - Choreography Activities MUST NOT be included as a flowElement for a Process.  
  - Activities, Data Associations, and Data Objects MUST NOT be included as a flowElement for a Choreography. |
| laneSets: LaneSet [0..*] | This attribute defines the list of LaneSets used in the FlowElementContainers. LaneSets are not used for Choreographies or Sub-Choreographies. |
8.3.9 Gateways

Gateways are used to control how the Process flows (how Tokens flow) through Sequence Flows as they converge and diverge within a Process. If the flow does not need to be controlled, then a Gateway is not needed. The term “gateway” implies that there is a gating mechanism that either allows or disallows passage through the Gateway—that is, as tokens arrive at a Gateway, they can be merged together on input and/or split apart on output as the Gateway mechanisms are invoked.

Gateways, like Activities, are capable of consuming or generating additional control tokens, effectively controlling the execution semantics of a given Process. The main difference is that Gateways do not represent ‘work’ being done and they are considered to have zero effect on the operational measures of the Process being executed (cost, time, etc.).

The Gateway controls the flow of both diverging and converging Sequence Flows. That is, a single Gateway could have multiple input and multiple output flows. Modelers and modeling tools might want to enforce a best practice of a Gateway only performing one of these functions. Thus, it would take two sequential Gateways to first converge and then to diverge the Sequence Flows.

Figure 8.24 - Gateway class diagram

The details for the types of Gateways (Exclusive, Inclusive, Parallel, Event-Based, and Complex) is defined on page 287 for Processes and on page 344 for Choreographies.
The **Gateway** class is an abstract type. Its concrete subclasses define the specific semantics of individual **Gateway** types, defining how the **Gateway** behaves in different situations.

The **Gateway** element inherits the attributes and model associations of **FlowElement** (see Table 8.44). Table 8.46 presents the additional attributes of the **Gateway** element.

### Table 8.46 – Gateway attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>gatewayDirection: GatewayDirection = Unspecified</td>
<td>An attribute that adds constraints on how the <strong>Gateway</strong> MAY be used.</td>
</tr>
<tr>
<td>{ Unspecified</td>
<td>Converging</td>
</tr>
<tr>
<td></td>
<td>• <strong>Converging</strong>: This <strong>Gateway</strong> MAY have multiple incoming <strong>Sequence Flows</strong> but MUST have no more than one outgoing <strong>Sequence Flow</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Diverging</strong>: This <strong>Gateway</strong> MAY have multiple outgoing <strong>Sequence Flows</strong> but MUST have no more than one incoming <strong>Sequence Flow</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Mixed</strong>: This <strong>Gateway</strong> contains multiple outgoing and multiple incoming <strong>Sequence Flows</strong>.</td>
</tr>
</tbody>
</table>

### 8.3.10 Item Definition

**BPMN** elements, such as **DataObjects** and **Messages**, represent items that are manipulated, transferred, transformed or stored during **Process** flows. These items can be either physical items, such as the mechanical part of a vehicle, or information items such as a catalog of the mechanical parts of a vehicle.

An important characteristics of items in **Process** is their structure. **BPMN** does not require a particular format for this data structure, but it does designate XML Schema as its default. The **structure** attribute references the actual data structure.

The default format of the data structure for all elements can be specified in the **Definitions** element using the **typeLanguage** attribute. For example, a **typeLanguage** value of [http://www.w3.org/2001/XMLSchema](http://www.w3.org/2001/XMLSchema) indicates that the data structures using by elements within that Definitions are in the form of XML Schema types. If unspecified, the default is XML schema. An Import is used to further identify the location of the data structure (if applicable). For example, in the case of data structures contributed by an XML schema, an Import would be used to specify the file location of that schema.

Structure definitions are always defined as separate entities, so they cannot be inlined in one of their usages. You will see that in every mention of structure definition there is a “reference” to the element. This is why this class inherits from **RootElement**.

An **ItemDefinition** element can specify an import reference where the proper definition of the structure is defined.
In cases where the data structure represents a collection, the multiplicity can be projected into the attribute `isCollection`. If this attribute is set to “true,” but the actual type is not a collection type, the model is considered as invalid. BPMN compliant tools might support an automatic check for these inconsistencies and report this as an error. The default value for this element is “false.”

The `itemKind` attribute specifies the nature of an item which can be a physical or an information item.

Figure 8.25 shows the ItemDefinition class diagram. When an ItemDefinition is defined it is contained in Definitions.

![ItemDefinition class diagram]

**Figure 8.25 - ItemDefinition class diagram**

The ItemDefinition element inherits the attributes and model associations `BaseElement` (see Table 8.5) through its relationship to `RootElement`. Table 8.47 presents the additional attributes and model associations for the ItemDefinition element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemKind: ItemKind = Information { Information</td>
<td>Physical }</td>
</tr>
<tr>
<td>structureRef: [Element [0..1]]</td>
<td>The concrete data structure to be used.</td>
</tr>
<tr>
<td>import: Import [0..1]</td>
<td>Identifies the location of the data structure and its format. If the importType attribute is left unspecified, the typeLanguage specified in the Definitions that contains this ItemDefinition is assumed.</td>
</tr>
<tr>
<td>isCollection: boolean = False</td>
<td>Setting this flag to true indicates that the actual data type is a collection.</td>
</tr>
</tbody>
</table>
8.3.11 Message

A **Message** represents the content of a communication between two *Participants*. In **BPMN 2.0**, a **Message** is a graphical decorator (it was a supporting element in **BPMN 1.2**). An **ItemDefinition** is used to specify the **Message** structure.

When displayed in a diagram:

- In a **Message** is a rectangle with converging diagonal lines in the upper half of the rectangle to give the appearance of an envelope (see Figure 8.26). It MUST be drawn with a single thin line.
- The use of text, color, size, and lines for a **Message** MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 41.

![Figure 8.26 - A Message](image)

In addition, when used in a **Choreography** Diagram more than one **Message** MAY be used for a single **Choreography Task**. In this case, it is important to know the first (initiating) **Message** of the interaction. For return (non-initiating) **Messages** the symbol of the **Message** is shaded with a light fill (see Figure 8.27).

![Figure 8.27 - A non-initiating Message](image)

- Any **Message** sent by the non-initiating **Participant** or **Sub-Choreography** MUST be shaded with a light fill.

In a **Collaboration**, the communication itself is represented by a **Message Flow** (see “Message Flow” below for more details). The **Message** can be optionally depicted as a graphical decorator on a **Message Flow** in a **Collaboration** (see Figure 8.28 and Figure 8.29).
In a Choreography, the communication is represented by a Choreography Task (see page 323). The Message can be depicted as a decorator with a Choreography Task in a Choreography (see Figure 8.29).

Figure 8.30 displays the class diagram showing the attributes and model associations for the Message element.
The Message element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to RootElement. Table 8.48 presents the additional attributes and model associations for the Message element.

Table 8.48 – Message attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>Name is a text description of the Message.</td>
</tr>
<tr>
<td>itemRef : ItemDefinition [0..1]</td>
<td>An ItemDefinition is used to define the “payload” of the Message.</td>
</tr>
</tbody>
</table>

8.3.12 Resources

The Resource class is used to specify resources that can be referenced by Activities. These Resources can be Human Resources as well as any other resource assigned to Activities during Process execution time.

The definition of a Resource is “abstract,” because it only defines the Resource, without detailing how e.g., actual user IDs are associated at runtime. Multiple Activities can utilize the same Resource.
Every Resource can define a set of ResourceParameters. These parameters can be used at runtime to define query e.g., into an Organizational Directory. Every Activity referencing a parameterized Resource can bind values available in the scope of the Activity to these parameters.

Figure 8.31 - Resource class diagram

The Resource element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to RootElement. Table 8.51 presents the additional model associations for the Resource element.

Table 8.49 – Resource attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>This attribute specifies the name of the Resource.</td>
</tr>
<tr>
<td>resourceParameters: ResourceParameter [0..*]</td>
<td>This model association specifies the definition of the parameters needed at runtime to resolve the Resource.</td>
</tr>
</tbody>
</table>

As mentioned before, the Resource can define a set of parameters to define a query to resolve the actual resources (e.g., user ids).

The ResourceParameter element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to RootElement. Table 8.51 presents the additional model associations for the ResourceParameter element.
8.3.13 Sequence Flow

A Sequence Flow is used to show the order of Flow Elements in a Process or a Choreography. Each Sequence Flow has only one source and only one target. The source and target MUST be from the set of the following Flow Elements: Events (Start, Intermediate, and End), Activities (Task and Sub-Process; for Processes), Choreography Activities (Choreography Task and Sub-Choreography; for Choreographies), and Gateways.

- A Sequence Flow is line with a solid arrowhead that MUST be drawn with a solid single line (as seen in Figure 8.32).
- The use of text, color, size, and lines for a Sequence Flow MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 41.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>Specifies the name of the query parameter.</td>
</tr>
<tr>
<td>type: ItemDefinition</td>
<td>Specifies the type of the query parameter.</td>
</tr>
<tr>
<td>isRequired: boolean</td>
<td>Specifies if a parameter is optional or mandatory.</td>
</tr>
</tbody>
</table>

Figure 8.32 - A Sequence Flow

A Sequence Flow can optionally define a condition Expression, indicating that the token will be passed down the Sequence Flow only if the Expression evaluates to true. This Expression is typically used when the source of the Sequence Flow is a Gateway or an Activity.

- A conditional outgoing Sequence Flow from an Activity MUST be drawn with a mini-diamond marker at the beginning of the connector (as seen in Figure 8.33).
- If a conditional Sequence Flow is used from a source Activity, then there MUST be at least one other outgoing Sequence Flow from that Activity.

- Conditional outgoing Sequence Flows from a Gateway MUST NOT be drawn with a mini-diamond marker at the beginning of the connector.
- A source Gateway MUST NOT be of type Parallel or Event.

Figure 8.33 - A Conditional Sequence Flow
A **Sequence Flow** that has an **Exclusive**, **Inclusive**, or **Complex Gateway** or an **Activity** as its source can also be defined with as **default**. Such a **Sequence Flow** will have a marker to show that it is a **default** flow. The **default Sequence Flow** is taken (a token is passed) only if all the other outgoing **Sequence Flows** from the **Activity** or **Gateway** are not valid (i.e., their condition **Expressions are false**).

- A **default outgoing Sequence Flow** **MUST** be drawn with a slash marker at the beginning of the connector (as seen in Figure 8.34).

---

**Figure 8.34 - A Default Sequence Flow**

![Sequence Flow class diagram](image)

**Figure 8.35 - SequenceFlow class diagram**

The **Sequence Flow** element inherits the attributes and model associations of **FlowElement** (see Table 8.44). Table 8.51 presents the additional attributes and model associations of the **Sequence Flow** element.
Flow Node

The FlowNode element is used to provide a single element as the source and target Sequence Flow associations (see Figure 8.35) instead of the individual associations of the elements that can connect to Sequence Flows (see the section above). Only the Gateway, Activity, Choreography Activity, and Event elements can connect to Sequence Flows and thus, these elements are the only ones that are sub-classes of FlowNode.

Since Gateway, Activity, Choreography Activity, and Event have their own attributes, model associations, and inheritances; the FlowNode element does not inherit from any other BPMN element. Table 8.52 presents the additional model associations of the FlowNode element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourceRef: FlowNode</td>
<td>The FlowNode that the Sequence Flow is connecting from. For a Process: Of the types of FlowNode, only Activities, Gateways, and Events can be the source. However, Activities that are Event Sub-Processes are not allowed to be a source. For a Choreography: Of the types of FlowNode, only Choreography Activities, Gateways, and Events can be the source.</td>
</tr>
<tr>
<td>targetRef: FlowNode</td>
<td>The FlowNode that the Sequence Flow is connecting to. For a Process: Of the types of FlowNode, only Activities, Gateways, and Events can be the target. However, Activities that are Event Sub-Processes are not allowed to be a target. For a Choreography: Of the types of FlowNode, only Choreography Activities, Gateways, and Events can be the target.</td>
</tr>
<tr>
<td>conditionExpression: Expression [0..1]</td>
<td>An optional boolean Expression that acts as a gating condition. A token will only be placed on this Sequence Flow if this conditionExpression evaluates to true.</td>
</tr>
</tbody>
</table>
| isImmediate: boolean [0..1] | An optional boolean value specifying whether Activities or Choreography Activities not in the model containing the Sequence Flow can occur between the elements connected by the Sequence Flow. If the value is false, they MAY NOT occur. If the value is true, they MAY occur. Also see the isClosed attribute on Process, Choreography, and Collaboration. When the attribute has no value, the default semantics depends on the kind of model containing Sequence Flows:  
  - For non-executable Processes (public Processes and non-executable private Processes) and Choreographies no value has the same semantics as if the value were false.  
  - For an executable Processes no value has the same semantics as if the value were true.  
  - For executable Processes, the attribute MUST NOT be false. |
### 8.3.14 Common Package XML Schemas

#### Table 8.53 – Error XML schema

```xml
<xsd:element name="error" type="tError" substitutionGroup="rootElement"/>
<xsd:complexType name="tError">
  <xsd:complexContent>
    <xsd:extension base="tRootElement">
      <xsd:attribute name="name" type="xsd:string"/>
      <xsd:attribute name="errorCode" type="xsd:string"/>
      <xsd:attribute name="structureRef" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

#### Table 8.54 – Escalation XML schema

```xml
<xsd:element name="escalation" type="tEscalation" substitutionGroup="rootElement"/>
<xsd:complexType name="tEscalation">
  <xsd:complexContent>
    <xsd:extension base="tRootElement">
      <xsd:attribute name="name" type="xsd:string"/>
      <xsd:attribute name="escalationCode" type="xsd:string"/>
      <xsd:attribute name="structureRef" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

#### Table 8.55 – Expression XML schema

```xml
<xsd:element name="expression" type="tExpression"/>
<xsd:complexType name="tExpression">
  <xsd:complexContent>
    <xsd:extension base="tBaseElementWithMixedContent"/>
  </xsd:complexContent>
</xsd:complexType>
```
Table 8.56 – FlowElement XML schema

```xml
<xsd:element name="flowElement" type="tFlowElement"/>
<xsd:complexType name="tFlowElement" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:sequence>
        <xsd:element ref="auditing" minOccurs="0" maxOccurs="1"/>
        <xsd:element ref="monitoring" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="categoryValueRef" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 8.57 – FlowNode XML schema

```xml
<xsd:element name="flowNode" type="tFlowNode"/>
<xsd:complexType name="tFlowNode" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="tFlowElement">
      <xsd:sequence>
        <xsd:element name="incoming" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="outgoing" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 8.58 – FormalExpression XML schema

```xml
<xsd:element name="formalExpression" type="tFormalExpression" substitutionGroup="expression"/>
<xsd:complexType name="tFormalExpression">
  <xsd:complexContent>
    <xsd:extension base="tExpression">
      <xsd:attribute name="language" type="xsd:anyURI" use="optional"/>
      <xsd:attribute name="evaluatesToTypeRef" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
Table 8.59 – InputOutputBinding XML schema
<xsd:element name="ioBinding" type="tinputOutputBinding"/>
<xsd:complexType name="tinputOutputBinding">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:attribute name="inputDataRef" type="xsd:IDREF"/>
      <xsd:attribute name="outputDataRef" type="xsd:IDREF"/>
      <xsd:attribute name="operationRef" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

Table 8.60 – ItemDefinition XML schema
<xsd:element name="itemDefinition" type="tItemDefinition" substitutionGroup="RootElement"/>
<xsd:complexType name="tItemDefinition">
  <xsd:complexContent>
    <xsd:extension base="tRootElement">
      <xsd:attribute name="structureRef" type="xsd:QName"/>
      <xsd:attribute name="isCollection" type="xsd:boolean" default="false"/>
      <xsd:attribute name="itemKind" type="tItemKind" default="Information"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="tItemKind">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Information"/>
    <xsd:enumeration value="Physical"/>
  </xsd:restriction>
</xsd:simpleType>

Table 8.61 – Message XML schema
<xsd:element name="message" type="tMessage" substitutionGroup="RootElement"/>
<xsd:complexType name="tMessage">
  <xsd:complexContent>
    <xsd:extension base="tRootElement">
      <xsd:attribute name="name" type="xsd:string"/>
      <xsd:attribute name="itemRef" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

Table 8.62 – Resources XML schema
<xsd:element name="resource" type="tResource" substitutionGroup="RootElement"/>
<xsd:complexType name="tResource">
    <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:sequence>
                <xsd:element ref="resourceParameter" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string" use="required"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

Table 8.63 – ResourceParameter XML schema

<xsd:element name="resourceParameter" type="tResourceParameter"/>
<xsd:complexType name="tResourceParameter">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:attribute name="name" type="xsd:string"/>
            <xsd:attribute name="type" type="xsd:QName"/>
            <xsd:attribute name="isRequired" type="xsd:Boolean"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

Table 8.64 – SequenceFlow XML schema

<xsd:element name="sequenceFlow" type="tSequenceFlow" substitutionGroup="flowElement"/>
<xsd:complexType name="tSequenceFlow">
    <xsd:complexContent>
        <xsd:extension base="tFlowElement">
            <xsd:sequence>
                <xsd:element name="conditionExpression" type="tExpression" minOccurs="0" maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="sourceRef" type="xsd:IDREF" use="required"/>
            <xsd:attribute name="targetRef" type="xsd:IDREF" use="required"/>
            <xsd:attribute name="isImmediate" type="xsd:boolean" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
8.4 Services

The Service package contains constructs necessary for modeling services, interfaces, and operations.

![Service class diagram](image)

**Figure 8.36 - The Service class diagram**

8.4.1 Interface

An Interface defines a set of operations that are implemented by Services.

The Interface inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship toRootElement. Table 8.65 presents the additional attributes and model associations of the Interface.
8.4.2 EndPoint

The actual definition of the service address is out of scope of BPMN 2.0. The EndPoint element is an extension point and extends from RootElement. The EndPoint element MAY be extended with endpoint reference definitions introduced in other specifications (e.g., WS-Addressing).

EndPoints can be specified for Participants.

8.4.3 Operation

An Operation defines Messages that are consumed and, optionally, produced when the Operation is called. It can also define zero or more errors that are returned when operation fails. The Operation inherits the attributes and model associations of BaseElement (see Table 8.5). Table 8.66 below presents the additional attributes and model associations of the Operation.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>The descriptive name of the element.</td>
</tr>
<tr>
<td>operations: Operation [1..*]</td>
<td>This attribute specifies operations that are defined as part of the Interface. An Interface has at least one Operation.</td>
</tr>
<tr>
<td>callableElements: CallableElement [0..*]</td>
<td>The CallableElements that use this Interface.</td>
</tr>
<tr>
<td>implementationRef: Element [0..1]</td>
<td>This attribute allows to reference a concrete artifact in the underlying implementation technology representing that interface, such as a WSDL porttype.</td>
</tr>
</tbody>
</table>
8.4.4 Service Package XML Schemas

Table 8.67 – Interface XML schema

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>The descriptive name of the element.</td>
</tr>
<tr>
<td>inMessageRef: Message</td>
<td>This attribute specifies the input Message of the Operation. An Operation has exactly one input Message.</td>
</tr>
<tr>
<td>outMessageRef: Message [0..1]</td>
<td>This attribute specifies the output Message of the Operation. An Operation has at most one input Message.</td>
</tr>
<tr>
<td>errorRef: Error [0..*]</td>
<td>This attribute specifies errors that the Operation may return. An Operation MAY refer to zero or more Error elements.</td>
</tr>
<tr>
<td>implementationRef: Element [0..1]</td>
<td>This attribute allows to reference a concrete artifact in the underlying implementation technology representing that operation, such as a WSDL operation.</td>
</tr>
</tbody>
</table>

Table 8.68 – Operation XML schema

```xml
<xsd:element name="operation" type="tOperation"/>
<xsd:complexType name="tOperation">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:sequence>
        <xsd:element name="inMessageRef" type="xsd:QName" minOccurs="1" maxOccurs="1"/>
        <xsd:element name="outMessageRef" type="xsd:QName" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="errorRef" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexType>
```

Table 8.66 – Operation attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>The descriptive name of the element.</td>
</tr>
<tr>
<td>inMessageRef: Message</td>
<td>This attribute specifies the input Message of the Operation. An Operation has exactly one input Message.</td>
</tr>
<tr>
<td>outMessageRef: Message [0..1]</td>
<td>This attribute specifies the output Message of the Operation. An Operation has at most one input Message.</td>
</tr>
<tr>
<td>errorRef: Error [0..*]</td>
<td>This attribute specifies errors that the Operation may return. An Operation MAY refer to zero or more Error elements.</td>
</tr>
<tr>
<td>implementationRef: Element [0..1]</td>
<td>This attribute allows to reference a concrete artifact in the underlying implementation technology representing that operation, such as a WSDL operation.</td>
</tr>
</tbody>
</table>
Table 8.69 – EndPoint XML schema

```xml
<xsd:element name="endPoint" type="tEndPoint"/>
<xsd:complexType name="tEndPoint">
    <xsd:complexContent>
        <xsd:extension base="tRootElement"/>
    </xsd:complexContent>
</xsd:complexType>
```
9 Collaboration

Note – The contents of this chapter are REQUIRED for BPMN Choreography Modeling Conformance, BPMN Process Modeling Conformance, or for BPMN Complete Conformance. However, this chapter is NOT REQUIRED for BPMN Process Execution Conformance or BPMN BPEL Process Execution Conformance. For more information about BPMN conformance types, see page 2.

The Collaboration package contains classes that are used for modeling Collaborations, which is a collection of Participants shown as Pools, their interactions as shown by Message Flows, and MAY include Processes within the Pools and/or Choreographies between the Pools (see Figure 9.1). A Choreography is an extended type of Collaboration. When a Collaboration is defined it is contained in Definitions.

Figure 9.1 - Classes in the Collaboration package
The **Collaboration** element inherits the attributes and model associations of **BaseElement** (see Table 8.5) through its relationship to **RootElement**. Table 9.1 presents the additional attributes and model associations for the **Collaboration** element.

### Table 9.1 - Collaboration Attributes and Model Associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>Name is a text description of the Collaboration.</td>
</tr>
<tr>
<td>choreographyRef: Choreography [0..*]</td>
<td>The choreographyRef model association defines the Choreographies that can be shown between the Pools of the Collaboration. A Choreography specifies a business contract (or the order in which messages will be exchanged) between interacting Participants. See page 315 for more details on Choreography. The participantAssociations (see below) are used to map the Participants of the Choreography to the Participants of the Collaboration. The MessageFlowAssociations (see below) are used to map the Message Flows of the Choreography to the Message Flows of the Collaboration. The conversationAssociations (see below) are used to map the Conversations of the Choreography to the Conversations of the Collaboration. Note that this attribute is not applicable for Choreography or GlobalConversation which are a subtypes of Collaboration. Thus, a Choreography cannot reference another Choreography.</td>
</tr>
<tr>
<td>correlationKeys: CorrelationKey [0..*]</td>
<td>This association specifies CorrelationKeys used to associate Messages to a particular Collaboration.</td>
</tr>
</tbody>
</table>
| conversationAssociations: ConversationAssociation [0..*] | This attribute provides a list of mappings from the Conversations of a referenced Collaboration to the Conversations of another Collaboration. It is used when:  
  - When a Choreography is referenced by a Collaboration. |
| conversations: ConversationNode [0..*] | The conversations model aggregation relationship allows a Collaboration to contain Conversation elements, in order to group Message Flows of the Collaboration and associate correlation information, as is REQUIRED for the definition of a Process model. The Conversation elements will be visualized if the Collaboration is a Collaboration, but not for a Choreography. |
| conversationLinks: ConversationLink [0..*] | This provides the Conversation Links that are used in the Collaboration. |
| artifacts: Artifact [0..*] | This attribute provides the list of Artifacts that are contained within the Collaboration. |
A set of Messages Flow of a particular Collaboration MAY belong to a Conversation. A Conversation is a set of Message Flows that share a particular purpose—i.e., they all relate to the handling of a single order (see page 124 for more information about Conversations).

### 9.1 Basic Collaboration Concepts

A Collaboration usually contains two or more Pools, representing the Participants in the Collaboration. The Message exchange between the Participants is shown by a Message Flow that connects two Pools (or the objects within the Pools). The Messages associated with the Message Flows MAY also be shown. See Figure 9.3, Figure 9.4, and Figure 9.5 for examples of Collaborations.

A Pool MAY be empty, a “black box,” or MAY show a Process within. Choreographies MAY be shown “in between” the Pools as they bisect the Message Flows between the Pools. All combinations of Pools, Processes, and a Choreography are allowed in a Collaboration.

<table>
<thead>
<tr>
<th><strong>participants</strong>: Participant [0..*]</th>
<th>This provides the list of Participants that are used in the Collaboration. Participants are visualized as Pools in a Collaboration and as Participant Bands in Choreography Activities in a Choreography.</th>
</tr>
</thead>
</table>
| **participantAssociations**: ParticipantAssociations [0..*] | This attribute provides a list of mappings from the Participants of a referenced Collaboration to the Participants of another Collaboration. It is used in the following situations:  
  * When a Choreography is referenced by the Collaboration.  
  * When a definitional Collaboration for a Process is referenced through a Call Activity (and mapped to definitional Collaboration of the calling Process). |
| **messageFlow**: Message Flow [0..*] | This provides the list of Message Flows that are used in the Collaboration. Message Flows are visualized in Collaboration (as dashed line) and hidden in Choreography. |
| **messageFlowAssociations**: Message Flow Association [0..*] | This attribute provides a list of mappings for the Message Flows of the Collaboration to Message Flows of a referenced model. It is used in the following situation:  
  * When a Choreography is referenced by a Collaboration. This allows the “wiring up” of the Collaboration Message Flows to the appropriate Choreography Activities. |
| **IsClosed**: boolean = false | A boolean value specifying whether Message Flows not modeled in the Collaboration can occur when the Collaboration is carried out.  
  * If the value is true, they MAY NOT occur.  
  * If the value is false, they MAY occur. |
9.1.1 Use of BPMN Common Elements

Some BPMN elements are common to both Process and Choreography, as well as Collaboration; they are used in these diagrams. The next few sections will describe the use of Messages, Message Flows, Participants, Sequence Flows, Artifacts, Correlations, Expressions, and Services in Choreography.

9.2 Pool and Participant

A Pool is the graphical representation of a Participant in a Collaboration. A Participant (see page 114) can be a specific PartnerEntity (e.g., a company) or can be a more general PartnerRole (e.g., a buyer, seller, or manufacturer). A Pool MAY or MAY NOT reference a Process. A Pool is NOT REQUIRED to contain a Process, i.e., it can be a “black box.”

- A Pool is a square-cornered rectangle that MUST be drawn with a solid single line (see Figure 9.2).
- The label for the Pool MAY be placed in any location and direction within the Pool, but MUST be separated from the contents of the Pool by a single line.
- If the Pool is a black box (i.e., does not contain a Process), then the label for the Pool MAY be placed anywhere within the Pool without a single line separator.
- One, and only one, Pool in a diagram MAY be presented without a boundary. If there is more than one Pool in the diagram, then the remaining Pools MUST have a boundary.

The use of text, color, size, and lines for a Pool MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 41.

![Figure 9.2 - A Pool](image)

To help with the clarity of the Diagram, a Pool extends the entire length of the Diagram, either horizontally or vertically. However, there is no specific restriction to the size and/or positioning of a Pool. Modelers and modeling tools can use Pools in a flexible manner in the interest of conserving the “real estate” of a Diagram on a screen or a printed page.

A Pool acts as the container for the Sequence Flows between Activities (of a contained Process). The Sequence Flows can cross the boundaries between Lanes of a Pool (see page 305 for more details on Lanes), but cannot cross the boundaries of a Pool. That is, a Process is fully contained within the Pool. The interaction between Pools is shown through Message Flows.

Another aspect of Pools is whether or not there is any Activity detailed within the Pool. Thus, a given Pool MAY be shown as a “White Box,” with all details (e.g., a Process) exposed, or as a “Black Box,” with all details hidden. No Sequence Flows are associated with a “Black Box” Pool, but Message Flows can attach to its boundaries (see Figure 9.3).
For a “White Box” Pool, the Activities within are organized by Sequence Flows. Message Flows can cross the Pool boundary to attach to the appropriate Activity (see Figure 9.4).

A Collaboration can contain two (2) or more Pools (i.e., Participants). However, a Process that represents the work performed from the point of view of the modeler or the modeler’s organization can be considered “internal” and is NOT REQUIRED to be surrounded by the boundary of the Pool, while the other Pools in the Diagram MUST have their boundary (see Figure 9.5).
BPMN specifies a marker for Pools: a multi-instance marker May be displayed for a Pool (see Figure 9.6). The marker is used if the Participant defined for the Pool is a multi-instance Participant. See page 117 for more information on Participant multiplicity.

- The marker for a Pool that is a multi-instance MUST be a set of three vertical lines in parallel.
- The marker, if used, MUST be centered at the bottom of the shape.

**Figure 9.6 - Pools with a Multi-Instance Participant Markers**

### 9.2.1 Participants

A Participant represents a specific PartnerEntity (e.g., a company) and/or a more general PartnerRole (e.g., a buyer, seller, or manufacturer) that are Participants in a Collaboration. A Participant is often responsible for the execution of the Process enclosed in a Pool; however, a Pool MAY be defined without a Process.

Figure 9.7 displays the class diagram of the Participant and its relationships to other BPMN elements. When Participants are defined they are contained within a Collaboration, which includes the sub-types of Choreography, GlobalConversation, or GlobalChoreographyTask.
Figure 9.7 - The Participant Class Diagram

The Participant element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 9.2 presents the additional attributes and model associations for the Participant element.
PartnerEntity

A PartnerEntity is one of the possible types of Participant (see the section above).

The PartnerEntity element inherits the attributes and model associations of BaseElement (see Figure 8.5). Table 9.3 presents the additional attributes and model associations for the PartnerEntity element.

Table 9.3 – PartnerEntity attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>Name is a text description of the PartnerEntity.</td>
</tr>
<tr>
<td>participantRef:</td>
<td>Specifies how the PartnerEntity participates in Collaborations and Choreographies.</td>
</tr>
</tbody>
</table>

Table 9.2 – Participant attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>Name is a text description of the Participant. The name of the Participant can be displayed directly or it can be substituted by the associated PartnerRole or PartnerEntity. Potentially, both the PartnerEntity name and PartnerRole name can be displayed for the Participant.</td>
</tr>
<tr>
<td>processRef: Process</td>
<td>The processRef attribute identifies the Process that the Participant uses in the Collaboration. The Process will be displayed within the Participant’s Pool.</td>
</tr>
<tr>
<td>partnerRoleRef:</td>
<td>The partnerRoleRef attribute identifies a PartnerRole that the Participant plays in the Collaboration. Both a PartnerRole and a PartnerEntity MAY be defined for the Participant. This attribute is derived from the participantRefs of PartnerRole.</td>
</tr>
<tr>
<td>partnerEntityRef:</td>
<td>The partnerEntityRef attribute identifies a PartnerEntity that the Participant plays in the Collaboration. Both a PartnerRole and a PartnerEntity MAY be defined for the Participant. This attribute is derived from the participantRefs of PartnerEntity.</td>
</tr>
<tr>
<td>interfaceRef: Interface</td>
<td>This association defines Interfaces that a Participant supports. The definition of Interfaces is provided on page 104.</td>
</tr>
<tr>
<td>participantMultiplicity: participant-Multiplicity</td>
<td>The participantMultiplicityRef model association is used to define Participants that represent more than one instance of the Participant for a given interaction. See the next section for more details on ParticipantMultiplicity.</td>
</tr>
<tr>
<td>endPointRefs: EndPoint</td>
<td>This attribute is used to specify the address (or endpoint reference) of concrete services realizing the Participant.</td>
</tr>
</tbody>
</table>
**PartnerRole**

A PartnerRole is one of the possible types of Participant (see the section above).

The PartnerRole element inherits the attributes and model associations of BaseElement (see Figure 8.5). Table 9.4 presents the additional attributes and model associations for the PartnerRole element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>Name is a text description of the PartnerRole.</td>
</tr>
<tr>
<td>participantRef: Participant [0..*]</td>
<td>Specifies how the PartnerRole participates in Collaborations and Choreographies.</td>
</tr>
</tbody>
</table>

### Participant Multiplicity

ParticipantMultiplicity is used to define the multiplicity of a Participant.

For example, a manufacturer can request a quote from multiple suppliers in a Collaboration.

![Figure 9.8 - A Pool with a Multiple Participant](image)

The following figure shows the Participant class diagram.

![Figure 9.9 - The Participant Multiplicity class diagram](image)
The multi-instance marker will be displayed in bottom center of the Pool (Participant - see Figure 9.9, above), or the Participant Band of a Choreography Activity (see page 321), when the ParticipantMultiplicity is associated with the Participant, and the maximum attribute is either not set, or has a value of two or more.

Table 9.5 presents the attributes for the ParticipantMultiplicity element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>minimum: integer = 0</td>
<td>The minimum attribute defines minimum number of Participants that MUST be involved in the Collaboration. If a value is specified in the maximum attribute, it MUST be greater or equal to this minimum value.</td>
</tr>
<tr>
<td>maximum: integer [0..1] = 1</td>
<td>The maximum attribute defines maximum number of Participants that MAY be involved in the Collaboration. The value of maximum MUST be one or greater, AND MUST be equal or greater than the minimum value.</td>
</tr>
</tbody>
</table>

Table 9.6 presents the Instance attributes of the ParticipantMultiplicity element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>numParticipants: integer [0..1]</td>
<td>The current number of the multiplicity of the Participant for this Choreography or Collaboration Instance.</td>
</tr>
</tbody>
</table>

ParticipantAssociation

These elements are used to do mapping between two elements that both contain Participants. There are situations where the Participants in different diagrams can be defined differently because they were developed independently, but represent the same thing. The ParticipantAssociation provides the mechanism to match up the Participants.

A ParticipantAssociation is used when an (outer) diagram with Participants contains an (inner) diagram that also has Participants. There are four usages of ParticipantAssociation. It is used when:

1. A Collaboration references a Choreography for inclusion between the Collaboration’s Pools (Participants). The Participants of the Choreography (the inner diagram) need to be mapped to the Participants of the Collaboration (the outer diagram).

2. A Call Conversation references a Collaboration or GlobalConversation. Thus, the Participants of the Collaboration or GlobalConversation (the inner diagram) need to be mapped to the Participants referenced by the Call Conversation (the outer element). Each Call Conversation contains its own set of ParticipantAssociations.

3. A Call Choreography references a Choreography or GlobalChoreographyTask. Thus, the Participants of the Choreography or GlobalChoreographyTask (the inner diagram) need to be mapped to the Participants referenced by the Call Choreography (the outer element). Each Call Choreography contains its own set of ParticipantAssociations.
4. A Call Activity within a Process that has a definitional Collaboration references another Process that also has a definitional Collaboration. The Participants of the definitional Collaboration of the called Process (the inner diagram) need to be mapped to the Participants of the definitional Collaboration of the calling Process (the outer diagram).

A ParticipantAssociation can be owned by the outer diagram or one its elements. Figure 9.10 shows the class diagram for the ParticipantAssociation element.

![ParticipantAssociation class diagram](image-url)

The ParticipantAssociation element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 9.7 presents the additional model associations for the ParticipantAssociation element.
9.2.2 Lanes

A Lane is a sub-partition within a Process (often within a Pool) and will extend the entire length of the Process level, either vertically (see Figure 10.123) or horizontally (see Figure 10.124). See page 305 for more information on Lanes.

9.3 Message Flow

A Message Flow is used to show the flow of Messages between two Participants that are prepared to send and receive them.

- A Message Flow MUST connect two separate Pools. They connect either to the Pool boundary or to Flow Objects within the Pool boundary. They MUST NOT connect two objects within the same Pool.
- A Message Flow is a line with an open circle line start and an open arrowhead line end that MUST be drawn with a dashed single line (see Figure 9.11).
- The use of text, color, size, and lines for a Message Flow MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 41.

Figure 9.11 - A Message Flow

In Collaboration Diagrams (the view showing the Choreography Process Combined with Orchestration Processes), a Message Flow can be extended to show the Message that is passed from one Participant to another (see Figure 9.12).

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>innerParticipantRef: Participant</td>
<td>This attribute defines the Participant of the referenced element (e.g., a Choreography to be used in a Collaboration) that will be mapped to the parent element (e.g., the Collaboration).</td>
</tr>
<tr>
<td>outerParticipantRef: Participant</td>
<td>This attribute defines the Participant of the parent element (e.g., a Collaboration references a Choreography) that will be mapped to the referenced element (e.g., the Choreography).</td>
</tr>
</tbody>
</table>
If a **Choreography** is included in the **Collaboration**, then the **Message Flow** will “pass-through” a **Choreography Task** as it connects from one **Participant** to another (see Figure 9.13).

---

**Figure 9.12 - A Message Flow with an Attached Message**

**Figure 9.13 - A Message Flow passing through a Choreography Task**
Figure 9.14 displays the class diagram of a **Message Flow** and its relationships to other **BPMN** elements. When a **Message Flow** is defined it is contained either within a **Collaboration**, a **Choreography**, or a **GlobalChoreographyTask**.

---

**Figure 9.14 - The Message Flow Class Diagram**

The **Message Flow** element inherits the attributes and model associations of **BaseElement** (see Table 8.5). Table 9.8 presents the additional attributes and model associations for the **Message Flow** element.
9.3.1 Interaction Node

The InteractionNode element is used to provide a single element as the source and target Message Flow associations (see Figure 9.14, above) instead of the individual associations of the elements that can connect to Message Flows (see the section above). Only the Pool/Participant, Activity, and Event elements can connect to Message Flows. The InteractionNode element is also used to provide a single element for source and target of Conversation Links, see page 132.

The InteractionNode element does not have any attributes or model associations and does not inherit from any other BPMN element. Since Pools/Participants, Activities, and Events have their own attributes, model associations, and inheritances, additional attributes and model associations for the InteractionNode element are not necessary.

9.3.2 Message Flow Associations

These elements are used to do mapping between two elements that both contain Message Flows. The MessageFlowAssociation provides the mechanism to match up the Message Flows.

A MessageFlowAssociation is used when an (outer) diagram with Message Flows contains an (inner) diagram that also has Message Flows. It is used when:

- A Collaboration references a Choreography for inclusion between the Collaboration's Pools (Participants). The Message Flows of the Choreography (the inner diagram) need to be mapped to the Message Flows of the Collaboration (the outer diagram).

- A Collaboration references a Conversation that contains Message Flows. The Message Flows of the Conversation can serve as a partial requirement for the Collaboration. Thus, the Message Flows of the Conversation (the inner diagram) need to be mapped to the Message Flows of the Collaboration (the outer diagram).

- A Choreography references a Conversation that contains Message Flows. The Message Flows of the Conversation can serve as a partial requirement for the Choreography. Thus, the Message Flows of the Conversation (the inner diagram) need to be mapped to the Message Flows of the Choreography (the outer diagram).

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>Name is a text description of the Message Flow.</td>
</tr>
<tr>
<td>sourceRef: InteractionNode</td>
<td>The InteractionNode that the Message Flow is connecting from. Of the types of InteractionNode, only Pools/Participants, Activities, and Events can be the source of a Message Flow.</td>
</tr>
<tr>
<td>targetRef: InteractionNode</td>
<td>The InteractionNode that the Message Flow is connecting to. Of the types of InteractionNode, only Pools/Participants, Activities, and Events can be the target of a Message Flow.</td>
</tr>
<tr>
<td>messageRef: Message [0..1]</td>
<td>The messageRef model association defines the Message that is passed via the Message Flow (see page 93 for more details).</td>
</tr>
</tbody>
</table>
Figure 9.15 shows the class diagram for the `MessageFlowAssociation` element.

![MessageFlowAssociation class diagram](image)

The `MessageFlowAssociation` element inherits the attributes and model associations of `BaseElement` (see Table 8.5). Table 9.9 presents the additional model associations for the `MessageFlowAssociation` element.

**Table 9.9 – `MessageFlowAssociation` attributes and model associations**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>innerMessageFlowRef</code></td>
<td>This attribute defines the <strong>Message Flow</strong> of the referenced element (e.g., a <strong>Choreography</strong> to be used in a <strong>Collaboration</strong>) that will be mapped to the parent element (e.g., the <strong>Collaboration</strong>).</td>
</tr>
<tr>
<td><code>outerMessageFlowRef</code></td>
<td>This attribute defines the <strong>Message Flow</strong> of the parent element (e.g., a <strong>Collaboration</strong> references a <strong>Choreography</strong>) that will be mapped to the referenced element (e.g., the <strong>Choreography</strong>).</td>
</tr>
</tbody>
</table>

### 9.4 Conversations

The **Conversation** diagram is particular usage of and an informal description of a **Collaboration** diagram. In general, it is a simplified version of **Collaboration**, but **Conversation** diagrams do maintain all the features of a **Collaboration**. In particular, **Processes** can appear within the **Participants (Pools)** of **Conversation** diagrams, to show how **Conversation** and **Activities** are related.

The view includes two additional graphical elements that do not exist in other **BPMN** views:

1. **Conversation Node** elements (**Conversation**, **Sub-Conversation**, and **Call Conversation**)
2. A **Conversation Link**
A **Conversation** is a logical grouping of **Message** exchanges (**Message Flows**) that can share a **Correlation**. A **Conversation** is the logical relation of **Message** exchanges. The logical relation, in practice, often concerns a business object(s) of interest, e.g., “Order,” “Shipment and Delivery,” and “Invoice.” Hence, a **Conversation** is associated with a set of name-value pairs, or a **Correlation Key** (e.g., “Order Identifier,” “Delivery Identifier”), which is recorded in the **Messages** that are exchanged. In this way, a **Message** can be routed to the specific **Process instance** responsible for receiving and processing the **Message**.

Figure 9.16 shows a simple example of a **Conversation** diagram.

![Figure 9.16 - A Conversation diagram](image)

Figure 9.17 shows a variation of the example above where the **Conversation** node has been expanded into its component **Message Flows**. Note that the diagram looks the same as a simple **Collaboration** diagram (as in Figure 9.3, above).

![Figure 9.17 - A Conversation diagram where the Conversation is expanded into Message Flows](image)

**Message** exchanges are related to each other and reflect distinct business scenarios. The relation is sometimes simple, e.g., a request followed by a response (and can be described as part of a structural interface of a service, e.g., as a WSDL operation definition). However for commercial business transactions managed through **Business Processes**, the relation can be complex, involving long-running, reciprocal **Message** exchanges, and that could extend beyond bilateral to complex, multilateral **Collaborations**. For example, in logistics, stock replenishments involve the following types scenarios: creation of sales orders; assignment of carriers for shipments combining different sales orders; crossing customs/quarantine; processing payment and investigating exceptions.
In addition to an orchestration Process, Conversations are relevant to a Choreography. However, the Conversations are not visualized in a Choreography. The difference is that a Choreography provides a multi-party perspective of a Conversation. This is because the Message exchanges modeled using Choreography Activities concern multiple Participants, unlike an orchestration Process where the Message sending and receiving elements relate to one Participant only. Other than the difference in perspective, the notion of Conversation remains the same across Choreography and orchestration - and the Message exchanges of a Conversation will ultimately be executed through an orchestration Process.

Since Collaboration provides a top-down, design-time modeling perspective for Message exchanges and their Conversations, an abstracted view of the all Conversations pertaining to a domain being modeled is available through a Conversation diagram. A Conversation diagram, as depicted in Figure 9.18, shows Conversations (as hexagons) between Participants. This provides a “bird’s eye” perspective of the different Conversations which relate to the domain.

![Conversation diagram](image-url)

**Figure 9.18 - Conversation diagram depicting several conversations between Participants in a related domain**

Figure 9.18, above, depicts 13 distinct Conversations between collaborating Participants in a logistics domain. As examples, Retailer and Supplier are involved in a Delivery Negotiations Conversation, and Consignee converses with Retailer and Supplier through Delivery/Dispatch Plan and Shipment Schedule Conversations respectively. More than two participants MAY be involved in a Conversation, e.g., Consignee, Consolidator and Shipper in Detailed Shipment
Schedule. The association of Participants to a **Conversation** are constrained to indicate whether one or many of Participants are involved. For example, one instance of Retailer converses with one instance of Supplier for Deliver Negotiations. However, one instance of Shipper converses with multiple instances of Carrier (indicated by the multi-instance symbol of the Pool for Carrier) for Carrier Planning. Note, multiplicity in constraints of **Conversation** diagrams means one or more (not zero or more).

The behavior of different **Conversations** is modeled through separate **Choreographies**, detailing the **Message** exchange sequences. In practice, **Conversations** which are closely related could be combined in the same **Choreography** models – e.g., a **Message** exchange in the **Delivery Negotiation** leads to **Shipment Schedule**, **Delivery Planning** and **Delivery/Dispatch Conversations** and these could be combined together in the same **Choreography**. Alternatively, they could be separated in different models.

Figure 9.19 shows a subset of the larger **Conversation** diagram of Figure 9.18, above. Figure 9.20 and Figure 9.21 show the drill down into the “Delivery Negotiations” **Sub-Conversation**. This expands the **Conversation** with the **Message Flows**, providing a structural view of a **Conversation** without the “clutter” of sequencing details in the same diagram. Figure 9.19 also indicates the **Correlation Key** involved in the **Message Flows** of the **Conversation**. For example, **Order Id** is necessary for all **Messages** of **Message Flows** in Delivery Negotiation. In addition, some **Message Flows** also require **Variation Id** (for dealing with shipment variations on a per line item basis).

![Figure 9.19 - An example of a Sub-Conversation](image)

Figure 9.19 - An example of a Sub-Conversation

Figure 9.20 shows how the **Sub-Conversation** of Figure 9.19, above, is expanded into a set of **Message Flows** and a lower-level **Conversation**.

![Figure 9.20 - An example of a Sub-Conversation expanded to a Conversation and Message Flow](image)

Figure 9.20 - An example of a Sub-Conversation expanded to a Conversation and Message Flow
Figure 9.21 shows how the Conversation of Figure 9.20, above, is also expanded into a set of Message Flows, combined with the previous Message Flows. Note that the newly exposed Message Flows of the lower-level Conversation will be correlated by the CorrelationKey of both the lower-level Conversation (Variation Id) and the higher-level Sub-Conversations (Order Id).

Figure 9.21 - An example of a Sub-Conversation that is fully expanded

In Figure 9.19, above, a hierarchical structure of Conversations can be seen with one set of Message Flows occurring within another in a parent-child relationship. In particular, after Planned Order Variations (keyed on Order Id) at the parent, a number of Message Flows of the child follow till Retailer Order and Delivery Variations Ack (keyed on Variation Id and Order Id). The remaining Message Flows (keyed on Order Id) are at the parent level. The child Conversation, as such, is part of the parent Conversation. Nesting is indicated graphically on a Conversation symbol (by a “+”), indicating a Sub-Conversation or a Call Conversation calling a Collaboration. Nesting can go to an arbitrary number of levels.

A common dependency between Conversations is overlap. Overlap occurs when two or more Conversations have some Message exchanges in common but not others. As an example in Figure 9.18, above, a Message is sent as part of Detailed Shipment Schedule (keyed on Carrier Schedule Id) to trigger Delivery Monitoring (keyed on Shipment Id). During Delivery Monitoring, Message could be sent to Detailed Shipment Schedule (to request modifications when transportation exceptions occur).

Splits and joins are special types of overlap scenarios. A Conversation split arises when, as part of a Conversation, a message is exchanged between two or more Participants that at the same time spawns a new, distinct Conversation (either between the same set of Participants or another set). Additionally, no further Message exchanges are shared by the split Conversations as well as no subsequent merges of them occur. An example is Delivery Planning which leads to Carrier Planning and Special Cover. A Conversation join occurs when several Conversations are merged into one Conversation and no further Message exchanges occur in the original Conversations, i.e., these Conversations are finalized. The generalization of a split and join is a Conversation refactor where Conversations are split into parallel Conversations and then are merged at a later point in time.

9.4.1 Conversation Node

ConversationNode is the abstract super class for all elements that can comprise the Conversation elements of a Collaboration diagram, which are Conversation (see page 130), Sub-Conversation (see page 130), and Call Conversation (see page 131).
Figure 9.22 - Metamodel of ConversationNode Related Elements

ConversationNodes are linked to and from Participants using **Conversation Links** (see page 132).

The ConversationNode element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 9.10 presents the additional attributes and model associations for the ConversationNode element.
### Conversation

A *Conversation* is an atomic element for a *Conversation (Collaboration)* diagram. It represents a set of *Message Flows* grouped together based on a concept and/or a *CorrelationKey*. A *Conversation* will involve two or more *Participants*.

- A *Conversation* is a hexagon that MUST be drawn with a single thin line (see Figure 9.23).

![Figure 9.23 – A Communication element](image)

The *Conversation* element inherits the attributes and model associations of *ConversationNode* (see Table 9.10), but does not contain any additional attributes or model associations.

### Sub-Conversation

A *Sub-Conversation* is a *ConversationNode* that is a hierarchical division within the parent *Collaboration*. A *Sub-Conversation* is a graphical object within a *Collaboration*, but it also can be “opened up” to show the lower-level details of the *Conversation*, which consist of *Message Flows*, *Conversations*, and/or other *Sub-Conversations*. The *Sub-Conversation* shares the *Participants* of its parent *Conversation*.

- A *Sub-Conversation* is a hexagon that MUST be drawn with a single thin line (see Figure 9.24).
- The *Sub-Corveration* marker MUST be a small square with a plus sign (+) inside. The square MUST be positioned at the bottom center of the shape.

---

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string [0..1]</td>
<td>Name is a text description of the <em>ConversationNode</em> element.</td>
</tr>
<tr>
<td>participantRefs: Participant [2..*]</td>
<td>This provides the list of <em>Participants</em> that are used in the <em>ConversationNode</em> from the list provided by the <em>ConversationNode</em>'s parent <em>Conversation</em>. This reference is visualized through a <em>Conversation Link</em> (see page 132).</td>
</tr>
<tr>
<td>correlationKeys: CorrelationKey [0..*]</td>
<td>This is a list of the <em>ConversationNode</em>'s <em>CorrelationKeys</em>, which are used to group <em>Message Flows</em> for the <em>ConversationNode</em>.</td>
</tr>
</tbody>
</table>
The **Sub-Conversation** element inherits the attributes and model associations of **ConversationNode** (see Table 9.10). Table 9.11 presents the additional model associations for the **Sub-Conversation** element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>conversationNodes:</td>
<td></td>
</tr>
<tr>
<td>ConversationNode [0..*]</td>
<td>The ConversationNodes model aggregation relationship allows a Sub-Conversation to contain other ConversationNodes, in order to group Message Flows of the Sub-Conversation and associate correlation information.</td>
</tr>
</tbody>
</table>

### 9.4.4 Call Conversation

A **Call Conversation** identifies a place in the **Conversation** (**Collaboration**) where a global **Conversation** or a **GlobalConversation** is used.

- If the **Call Conversation** calls a **GlobalConversation**, then the shape will be the same as a **Conversation**, but the boundary of the shape will **MUST** have a thick line (see Figure 9.25).
- If the **Call Conversation** calls a **Collaboration**, then the shape will be the same as a **Sub-Conversation**, but the boundary of the shape will **MUST** have a thick line (see Figure 9.26).
Note - The ConversationNode attribute messageFlowRef doesn’t apply to Call Conversations.

### 9.4.5 Global Conversation

A GlobalConversation is a reusable, atomic Conversation definition that can be called from within any Collaboration by a Call Conversation.

The GlobalConversation element inherits the attributes and model associations and Collaboration (see Table 9.1), but does not have any additional attributes or model associations.

A GlobalConversation is a restricted type of Collaboration, it is an “empty Collaboration.”

- A GlobalConversation MUST NOT contain any ConversationNodes.

Since a GlobalConversation does not have any Flow Elements, it does not require MessageFlowAssociations, ParticipantAssociations, or ConversationAssociations or Artifacts. It is basically a set of Participants, Message Flows, and CorrelationKeys intended for reuse. Also, the Collaboration attribute choreographyRef is not applicable to GlobalConversation.

### 9.4.6 Conversation Link

Conversation Links are used to connect ConversationNodes to and from Participants (Pools -- see Figure 9.27).

- Conversation Links MUST be drawn with double thin lines.

![Figure 9.27 – A Conversation Link element](image-url)
Processes can appear in the Participants (Pools) of Conversation diagrams, as shown in Figure 9.28. The invoicing and ordering Conversations have links into Activities and Events of the Process in the Order Processor. The other two Conversations do not have their links “expanded.” Conversation Links into Activities that are not Send or Receive Tasks indicate that the Activity will send or receive Messages of the Conversation at some level of nesting.

Figure 9.28 – Conversation links to Activities and Events
The **Conversation Link** element inherits the attributes and model associations of **BaseElement** (see Table 8.5). Table 9.13 presents the additional attributes and model associations for the **Conversation Link** element.

**Table 9.13 — Conversation Link Attributes and Model Associations**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string [0..1]</td>
<td>This attribute specifies the name of the Conversation Link.</td>
</tr>
<tr>
<td>sourceRef: InteractionNode</td>
<td>The InteractionNode that the Conversation Link is connecting from. A Conversation Link MUST connect to exactly one ConversationNode. If the sourceRef is not a ConversationNode, then the targetRef MUST be a ConversationNode.</td>
</tr>
<tr>
<td>targetRef: InteractionNode</td>
<td>The InteractionNode that the Conversation Link is connecting to. A Conversation Link MUST connect to exactly one ConversationNode. If the targetRef is not a ConversationNode, then the sourceRef MUST be a ConversationNode.</td>
</tr>
</tbody>
</table>

**Conversation Links** for Call Conversations show the names of Participants in nested Collaboration or global Collaborations, as identified by ParticipantAssociations. For example, Figure 9.30 has a Collaboration on the left with a Call Conversations to a Collaboration on the right. The Conversation Links on the left indicate
which *Participants* in the called *Collaboration* on the right correspond to which *Participants* in the calling *Collaboration* on the left. For example, the Credit Agency *Participants* on the right corresponds to the Financial Company *Participant* on the left. Participant Associations (not shown) tie each *Participant* in the *Collaboration* on the left to a *Participant* in the *Collaboration* on the right. They can be used to show the names of *Participants* in nested *Collaboration* or global *Collaborations*.

![Diagram showing call conversation links](image)

**Figure 9.30 – Call Conversation Links**

### 9.4.7 Conversation Association

A *[ConversationAssociation](#)* is used within *Collaborations* and *Choreographies* to apply a reusable *Conversation* to the *Message Flows* of those diagrams.

A *[ConversationAssociation](#)* is used when a diagram references a *Conversation* to provide *Message* correlation information and/or to logically group *Message Flows*. It is used when:

- A *Collaboration* references a *Choreography* for inclusion between the *Collaboration*’s *Pools* (*Participants*). The ConversationNodes of the *Choreography* (the inner diagram) need to be mapped to the ConversationNodes of the *Collaboration* (the outer diagram).
The `ConversationAssociation` element inherits the attributes and model associations of `BaseElement` (see Table 8.5). Table 9.14 presents the additional model associations for the `ConversationAssociation` element.

Table 9.14 – `ConversationAssociation` Model Associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>innerConversationNodeRef:</code></td>
<td>This attribute defines the <code>ConversationNodes</code> of the referenced element (e.g., a <code>Choreography</code> to be used in a <code>Collaboration</code>) that will be mapped to the parent element (e.g., the <code>Collaboration</code>).</td>
</tr>
<tr>
<td><code>outerConversationNodeRef:</code></td>
<td>This attribute defines the <code>ConversationNodes</code> of the parent element (e.g., a <code>Collaboration</code> references a <code>Choreography</code>) that will be mapped to the referenced element (e.g., the <code>Choreography</code>).</td>
</tr>
</tbody>
</table>

### 9.4.8 Correlations

Correlations are the mechanism that is used to assign the `Messages` to the proper `Process instance`, and can be defined for the `Message Flows` that belong to the `Conversation`. Correlations can be used to specify `Conversations` between `Processes` that follow a fairly simple `Conversation` pattern in the sense that:

- The conceptual data of the `Conversation` is well known and defined by the participating `Processes`. However this doesn’t mandate that underlying type systems are identical. It is sufficient that the data is known “conceptually” on a (potentially very high) business level.
- A `Conversation` takes place by means of simple `Message` exchange between `Processes`, no additional agreements MUST be considered.
- There exists send and receive `Tasks` accepting the conceptual data of the `Conversation`. (An Order send by a `Task` of a `Process` should be received by at least one `Task` of the participating `Process`).
• The correlation itself is defined in terms of correlation fields, which denote a subset of the conceptual data that should be used for the correlation. (For example, if the conceptual data comprises an order, then the correlation field might be denoted by the order ID).

In some applications it is useful to allow more Messages to be sent between Participants when a Collaboration is carried out than are contained in the Collaboration model. This enables Participants to exchange other Messages as needed without changing the Collaboration. If the isClosed attribute of a Collaboration has a value of false or no value, then Participants MAY send Messages to each other without additional Message Flows in the Collaboration. If the isClosed attribute of a Collaboration has a value of true, then Participants MAY NOT send Messages to each other without additional Message Flows in the Collaboration. If a Collaboration contains a Choreography, then the value of the isClosed attribute MUST be the same in both. Restrictions on unmodeled messaging specified with isClosed apply only under the Collaboration containing the restriction. PartnerEntities and PartnerRoles of the Participants MAY send Messages to each other under other Choreographies, Collaborations, and Conversations.

9.5 Process within Collaboration

Processes can be included in a Collaboration diagram. A Participant/Pool within the Collaboration can contain a Process (but they are NOT REQUIRED). An example of this is shown in Figure 9.4, above.

When a Lane (in a Process) represents a Conversation, the Flow Elements in the Lane (or elements nested or called in them) that send or receive Messages MUST do so as part of the Conversation represented by the Lane.

9.6 Choreography within Collaboration

Choreographies can be included in a Collaboration diagram. A Collaboration specifies how the Participants and Message Flows in the Choreography are matched up with the Participants and Message Flows in the Collaboration. A Collaboration uses ParticipantAssociations and MessageFlowAssociations for this purpose.

To handle the Participants, the innerParticipant of a ParticipantAssociation refers to a Participant in the Choreography, while the outerParticipant refers to a Participant in the Collaboration containing the Choreography. This mapping matches the Participant Bands of the Choreography Activities in the Choreography to the Pools in the Collaboration. Thus, the names in the Participant Bands are NOT REQUIRED (see Figure 9.32).
Figure 9.32 - An example of a Choreography within a Collaboration

To handle Message Flows, the innerMessageFlow of a MessageFlowAssociation refers to a Message Flow in the Choreography, while the outerMessageFlow refers to a Message Flow in the Collaboration containing the Choreography. This mapping matches the Message Flows of the Choreography (which are not visible) to the Message Flows in the Collaboration (which are visible). This allows the Message Flows of the Collaboration to be “wired up” through the appropriate Choreography Activity in the Choreography (see Figure 9.32).

The ParticipantAssociations might be derived from the partnerEntities or partnerRoles of the Participants. For example, if a Choreography Activity has a Participant with the same partnerEntity as a Participant in the Collaboration containing the Choreography, then these two Participants could be assumed to be the inner and outerParticipants of a ParticipantAssociation. Similarly, Message Flows that reference the same Message in a Call Choreography Activity and the Collaboration, could be automatically synchronized by a MessageFlowAssociation, if only one Message Flow has that Message.
Figure 9.33 - Choreography within Collaboration class diagram

9.7 Collaboration Package XML Schemas

Table 9.15 – Call Conversation XML schema

```xml
<xsd:element name="callConversation" type="tCallConversation" substitutionGroup="conversationNode"/>
<xsd:complexType name="tCallConversation">
  <xsd:complexContent>
    <xsd:extension base="tConversationNode">
      <xsd:sequence>
        <xsd:element ref="participantAssociation" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="calledCollaborationRef" type="xsd:QName" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
Table 9.16 – Collaboration XML schema

```xml
<xsd:element name="collaboration" type="tCollaboration" substitutionGroup="rootElement"/>
<xsd:complexType name="tCollaboration">
  <xsd:complexContent>
    <xsd:extension base="tRootElement">
      <xsd:sequence>
        <xsd:element name="choreography" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="participant" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="messageFlow" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="artifact" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="conversationNode" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="conversationLink" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="conversationAssociation" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="participantAssociation" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="correlationKey" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string"/>
      <xsd:attribute name="isClosed" type="xsd:boolean" default="false"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 9.17 – Conversation XML schema

```xml
<xsd:element name="conversation" type="tConversation" substitutionGroup="conversationNode"/>
<xsd:complexType name="tConversation">
  <xsd:complexContent>
    <xsd:extension base="tConversationNode"/>
  </xsd:complexContent>
</xsd:complexType>
```

Table 9.18 – ConversationAssociation XML schema

```xml
<xsd:element name="conversationAssociation" type="tConversationAssociation"/>
<xsd:complexType name="tConversationAssociation">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:attribute name="innerConversationNodeRef" type="xsd:QName" use="required"/>
      <xsd:attribute name="outerConversationNodeRef" type="xsd:QName" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 9.19 – ConversationAssociation XML schema

```xml
<xsd:element name="conversationLink" type="tConversationLink"/>
<xsd:complexType name="tConversationLink">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:attribute name="innerConversationNodeRef" type="xsd:QName" use="required"/>
      <xsd:attribute name="outerConversationNodeRef" type="xsd:QName" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
Table 9.20 – ConversationNode XML schema

```xml
<xsd:element name="conversation" type="tConversation" substitutionGroup="rootElement"/>
<xsd:complexType name="tConversation">
  <xsd:complexContent>
    <xsd:extension base="tCallableElement">
      <xsd:sequence>
        <xsd:element ref="conversationNode" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="participant" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="artifact" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="messageFlowRef" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="correlationKey" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="conversationRef" type="xsd:QName"/>
      <xsd:attribute name="correlationKeyRef" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 9.21 – Conversation Node XML schema

```xml
<xsd:element name="conversationNode" type="tConversationNode" abstract="true"/>
<xsd:complexType name="tConversationNode">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:sequence>
        <xsd:element name="messageFlowRef" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="participantRef" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="conversationRef" type="xsd:QName"/>
      <xsd:attribute name="correlationKeyRef" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 9.22 – Global Conversation XML schema

```xml
<xsd:element name="globalConversation" type="tGlobalConversation" substitutionGroup="collaboration"/>
<xsd:complexType name="tGlobalConversation">
```

Business Process Model and Notation, v2.0
Table 9.23 – MessageFlow XML schema

```xml
<xsd:element name="messageFlow" type="tMessageFlow"/>
<xsd:complexType name="tMessageFlow">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:attribute name="name" type="xsd:string" use="optional"/>
      <xsd:attribute name="sourceRef" type="xsd:QName" use="required"/>
      <xsd:attribute name="targetRef" type="xsd:QName" use="required"/>
      <xsd:attribute name="messageRef" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 9.24 – MessageFlowAssociation XML schema

```xml
<xsd:element name="messageFlowAssociation" type="tMessageFlowAssociation"/>
<xsd:complexType name="tMessageFlowAssociation">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:attribute name="innerMessageFlowRef" type="xsd:QName" use="required"/>
      <xsd:attribute name="outerMessageFlowRef" type="xsd:QName" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 9.25 – Participant XML schema

```xml
<xsd:element name="participant" type="tParticipant"/>
<xsd:complexType name="tParticipant">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:sequence>
        <xsd:element name="interfaceRef" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="endPointRef" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string"/>
      <xsd:attribute name="processRef" type="xsd:QName" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
Table 9.26 – ParticipantAssociation XML schema

```xml
<xsd:element name="participantAssociation" type="tParticipantAssociation"/>
<xsd:complexType name="tParticipantAssociation">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="innerParticipantRef" type="xsd:QName" use="required"/>
                <xsd:element name="outerParticipantRef" type="xsd:QName" use="required"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 9.27 – ParticipantMultiplicity XML schema

```xml
<xsd:element name="participantMultiplicity" type="tParticipantMultiplicity"/>
<xsd:complexType name="tParticipantMultiplicity">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:attribute name="minimum" type="xsd:int"/>
            <xsd:attribute name="maximum" type="xsd:int"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 9.28 – PartnerEntity XML schema

```xml
<xsd:element name="partnerEntity" type="tPartnerEntity" substitutionGroup="rootElement"/>
<xsd:complexType name="tPartnerEntity">
    <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:attribute name="name" type="xsd:string"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 9.29 – PartnerRole XML schema

```xml
<xsd:element name="partnerRole" type="tPartnerRole" substitutionGroup="rootElement"/>
<xsd:complexType name="tPartnerRole">
    <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:attribute name="name" type="xsd:string"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```
Table 9.30 – Sub-Conversation XML schema

```xml
<xsd:element name="subConversation" type="tSubConversation" substitutionGroup="conversationNode"/>
<xsd:complexType name="tSubConversation">
    <xsd:complexContent>
        <xsd:extension base="tConversationNode">
            <xsd:sequence>
                <xsd:element ref="conversationNode" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```
10 Process

**Note** – The content of this chapter is REQUIRED for BPMN Process Modeling Conformance or for BPMN Complete Conformance. However, this chapter is NOT REQUIRED for BPMN Process Choreography Conformance, BPMN Process Execution Conformance, or BPMN BPEL Process Execution Conformance. For more information about BPMN conformance types, see page 2.

A **Process** describes 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 Flows** that define finite execution semantics (see Figure 10.1). **Processes** can be defined at any level from enterprise-wide **Processes** to **Processes** performed by a single person. Low-level **Processes** can be grouped together to achieve a common business goal.

![Figure 10.1 - An Example of a Process](image)

Note that **BPMN** uses the term **Process** specifically to mean a set of *flow elements*. It uses the terms **Collaboration** and **Choreography** when modeling the interaction between **Processes**.

The **Process** package contains classes that are used for modeling the flow of **Activities**, **Events**, and **Gateways**, and how they are sequenced within a **Process** (see Figure 10.2). When a **Process** is defined it is contained within **Definitions**.
Figure 10.2 - Process class diagram

A Process is a CallableElement, allowing it to be referenced and reused by other Processes via the Call Activity construct. In this capacity, a Process MAY reference a set of Interfaces that define its external behavior.

A Process is a reusable element and can be imported and used within other Definitions.

Figure 10.3 shows the details of the attributes and model associations of a Process.
The Process element inherits the attributes and model associations of CallableElement (see Table 10.24) and of FlowElementContainer (see Table 8.45). Table 10.1 presents the additional attributes and model associations of the Process element:

### Table 10.1 – Process Attributes & Model Associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>processType: ProcessType = none { None</td>
<td>Private</td>
</tr>
</tbody>
</table>
An optional Boolean value specifying whether the **Process** is executable.

An executable **Process** is a private **Process** that has been modeled for the purpose of being executed according to the semantics of Chapter 14 (see page 426). Of course, during the development cycle of the **Process**, there will be stages where the **Process** does not have enough detail to be “executable.”

A non-executable **Process** is a private **Process** that has been modeled for the purpose of documenting **Process** behavior at a modeler-defined level of detail. Thus, information needed for execution, such as formal condition expressions are typically not included in a non-executable **Process**.

For public **Processes**, no value has the same semantics as if the value were *false*. The value MAY not be *true* for public **Processes**.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isExecutable: boolean [0..1]</td>
<td>An optional Boolean value specifying whether the <strong>Process</strong> is executable. An executable <strong>Process</strong> is a private <strong>Process</strong> that has been modeled for the purpose of being executed according to the semantics of Chapter 14 (see page 426). Of course, during the development cycle of the <strong>Process</strong>, there will be stages where the <strong>Process</strong> does not have enough detail to be “executable.” A non-executable <strong>Process</strong> is a private <strong>Process</strong> that has been modeled for the purpose of documenting <strong>Process</strong> behavior at a modeler-defined level of detail. Thus, information needed for execution, such as formal condition expressions are typically not included in a non-executable <strong>Process</strong>. For public <strong>Processes</strong>, no value has the same semantics as if the value were <em>false</em>. The value MAY not be <em>true</em> for public <strong>Processes</strong>.</td>
</tr>
<tr>
<td>auditing: Auditing [0..1]</td>
<td>This attribute provides a hook for specifying audit related properties.</td>
</tr>
<tr>
<td>monitoring: Monitoring [0..1]</td>
<td>This attribute provides a hook for specifying monitoring related properties.</td>
</tr>
<tr>
<td>artifacts: Artifact [0..*]</td>
<td>This attribute provides the list of <strong>Artifacts</strong> that are contained within the <strong>Process</strong>.</td>
</tr>
<tr>
<td>IsClosed: boolean = false</td>
<td>A boolean value specifying whether interactions, such as sending and receiving <strong>Messages</strong> and <strong>Events</strong>, not modeled in the <strong>Process</strong> can occur when the <strong>Process</strong> is executed or performed. If the value is <em>true</em>, they MAY NOT occur. If the value is <em>false</em>, they MAY occur.</td>
</tr>
<tr>
<td>supports: Process [0..*]</td>
<td>Modelers can declare that they intend all executions or performances of one <strong>Process</strong> to also be valid for another <strong>Process</strong>. This means they expect all the executions or performances of the first <strong>Processes</strong> to also follow the steps laid out in the second <strong>Process</strong>.</td>
</tr>
<tr>
<td>properties: Property [0..*]</td>
<td>Modeler-defined properties MAY be added to a <strong>Process</strong>. These properties are contained within the <strong>Process</strong>. All <strong>Tasks</strong> and <strong>Sub-Processes</strong> SHALL have access to these properties.</td>
</tr>
<tr>
<td>resources: ResourceRole [0..*]</td>
<td>Defines the resource that will perform or will be responsible for the <strong>Process</strong>. The resource, e.g., a performer, can be specified in the form of a specific individual, a group, an organization role or position, or an organization. Note that the assigned resources of the <strong>Process</strong> does not determine the assigned resources of the <strong>Activities</strong> that are contained by the <strong>Process</strong>. See more details about resource assignment on page 154.</td>
</tr>
</tbody>
</table>
In addition, a Process instance has attributes whose values MAY be referenced by Expressions (see Table 10.2). These values are only available when the Process is being executed.

Table 10.2 – Process instance attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>state: string = None</td>
<td>See Figure 13.2 (&quot;The Lifecycle of a BPMN Activity&quot;) in Section 13.2.2 for permissible values.</td>
</tr>
</tbody>
</table>

10.1 Basic Process Concepts

10.1.1 Types of BPMN Processes

Business Process modeling is used to communicate a wide variety of information to a wide variety of audiences. BPMN is designed to cover many types of modeling and allows the creation of end-to-end Business Processes. There are three basic types of BPMN Processes:

1. Private Non-executable (internal) Business Processes
2. Private Executable (internal) Business Processes
3. Public Processes

10.1.1.1 Private (Internal) Business Processes

Private Business Processes are those internal to a specific organization. These Processes have been generally called workflow or BPM Processes (see Figure 10.4). Another synonym typically used in the Web services area is the Orchestration of services. There are two types of private Processes: executable and non-executable. An executable Process is a Process that has been modeled for the purpose of being executed according to the semantics defined in Chapter 14 (see page 426). Of course, during the development cycle of the Process, there will be stages where the...
**Process** does not have enough detail to be “executable.” A non-executable **Process** is a private **Process** that has been modeled for the purpose of documenting **Process** behavior at a modeler-defined level of detail. Thus, information needed for execution, such as formal condition **Expressions** are typically not included in a non-executable **Process**.

If a swimlanes-like notation is used (e.g., a **Collaboration**, see below) then a private **Business Process** will be contained within a single **Pool**. The **Process** flow is therefore contained within the **Pool** and cannot cross the boundaries of the **Pool**. The flow of **Messages** can cross the **Pool** boundary to show the interactions that exist between separate private **Business Processes**.

**Figure 10.4 - Example of a private Business Process**

### 10.1.1.2 Public Processes

A public **Process** represents the interactions between a private **Business Process** and another **Process** or **Participant** (see Figure 10.5). Only those **Activities** that are used to communicate to the other **Participant(s)**, plus the order of these **Activities**, are included in the public **Process**. All other “internal” **Activities** of the private **Business Process** are not shown in the public **Process**. Thus, the public **Process** shows to the outside world the **Messages**, and the order of these **Messages**, that are needed to interact with that **Business Process**. **Public Processes** can be modeled separately or within a **Collaboration** to show the flow of **Messages** between the public **Process Activities** and other **Participants**. Note that the public type of **Process** was named “abstract” in BPMN 1.2.

**Figure 10.5 - Example of a public Process**

### 10.1.2 Use of BPMN Common Elements

Some **BPMN** elements are common to both **Process** and **Choreography**, as well as **Collaboration**; they are used in these diagrams. The next few sections will describe the use of **Messages**, **Message Flows**, **Participants**, **Sequence Flows**, **Artifacts**, **Correlations**, **Expressions**, and **Services** in **Choreography**.

The key graphical elements of **Gateways** and **Events** are also common to both **Choreography** and **Process**. Since their usage has a large impact, they are described in major sections of this chapter (see page 233 for **Events** and page 287 for **Gateways**).
10.2 Activities

An Activity is work that is performed within a Business Process. An Activity can be atomic or non-atomic (compound). The types of Activities that are a part of a Process are: Task, Sub-Process, and Call Activity, which allows the inclusion of re-usable Tasks and Processes in the diagram. However, a Process is not a specific graphical object. Instead, it is a set of graphical objects. The following sections will focus on the graphical objects Sub-Process and Task.

Activities represent points in a Process flow where work is performed. They are the executable elements of a BPMN Process.

The Activity class is an abstract element, sub-classing from FlowElement (as shown in Figure 10.6). Concrete sub-classes of Activity specify additional semantics above and beyond that defined for the generic Activity.

![Activity class diagram](image)

Figure 10.6 - Activity class diagram

The Activity class is the abstract super class for all concrete Activity types.

The Activity element inherits the attributes and model associations of FlowElement (see Table 8.44). Table 10.3 presents the additional attributes and model associations of the Activity element.
<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>isForCompensation: boolean = false</td>
<td>A flag that identifies whether this Activity is intended for the purposes of compensation. If false, then this Activity executes as a result of normal execution flow. If true, this Activity is only activated when a Compensation Event is detected and initiated under Compensation Event visibility scope (see page 281 for more information on scopes).</td>
</tr>
<tr>
<td>loopCharacteristics: LoopCharacteristics [0..1]</td>
<td>An Activity MAY be performed once or MAY be repeated. If repeated, the Activity MUST have loopCharacteristics that define the repetition criteria (if the isExecutable attribute of the Process is set to true).</td>
</tr>
<tr>
<td>resources: ResourceRole [0..*]</td>
<td>Defines the resource that will perform or will be responsible for the Activity. The resource, e.g., a performer, can be specified in the form of a specific individual, a group, an organization role or position, or an organization.</td>
</tr>
<tr>
<td>default: SequenceFlow [0..1]</td>
<td>The Sequence Flow that will receive a token when none of the conditionExpressions on other outgoing Sequence Flows evaluate to true. The default Sequence Flow should not have a conditionExpression. Any such Expression SHALL be ignored.</td>
</tr>
<tr>
<td>properties: Property [0..*]</td>
<td>Modeler-defined properties MAY be added to an Activity. These properties are contained within the Activity.</td>
</tr>
<tr>
<td>boundaryEventRefs: BoundaryEvent [0..*]</td>
<td>This references the Intermediate Events that are attached to the boundary of the Activity.</td>
</tr>
<tr>
<td>dataInputAssociations: DataInputAssociation [0..*]</td>
<td>An optional reference to the DataInputAssociations. A DataInputAssociation defines how the DataInput of the Activity’s InputOutputSpecification will be populated.</td>
</tr>
<tr>
<td>dataOutputAssociations: DataOutputAssociation [0..*]</td>
<td>An optional reference to the DataOutputAssociations.</td>
</tr>
<tr>
<td>startQuantity: integer = 1</td>
<td>The default value is 1. The value MUST NOT be less than 1. This attribute defines the number of tokens that MUST arrive before the Activity can begin. Note that any value for the attribute that is greater than 1 is an advanced type of modeling and should be used with caution.</td>
</tr>
</tbody>
</table>
In addition, an Activity instance has attributes whose values MAY be referenced by Expressions. These values are only available when the Activity is being executed.

Table 10.4 presents the instance attributes of the Activity element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>state: string = None</td>
<td>See Figure 13.2 (&quot;The Lifecycle of a BPMN Activity&quot;) in Section 13.2.2 for permissible values.</td>
</tr>
</tbody>
</table>

Sequence Flow Connections

See “Sequence Flow Connections Rules” on page 42 for the entire set of objects and how they MAY be sources or targets of Sequence Flows.

- An Activity MAY be a target for Sequence Flows; it can have multiple incoming Sequence Flows. Incoming Sequence Flows MAY be from an alternative path and/or parallel paths.

  - If the Activity does not have an incoming Sequence Flow, then the Activity MUST be instantiated when the Process is instantiated.

  - There are two exceptions to this: Compensation Activities and Event Sub-Processes.

Note – If the Activity has multiple incoming Sequence Flows, then this is considered uncontrolled flow. This means that when a token arrives from one of the Paths, the Activity will be instantiated. It will not wait for the arrival of tokens from the other paths. If another token arrives from the same path or another path, then a separate instance of the Activity will be created. If the flow needs to be controlled, then the flow should converge on a Gateway that precedes the Activities (see “Gateways” on page 287 for more information on Gateways).

- An Activity MAY be a source for Sequence Flows; it can have multiple outgoing Sequence Flows. If there are multiple outgoing Sequence Flows, then this means that a separate parallel path is being created for each Sequence Flow (i.e., tokens will be generated for each outgoing Sequence Flow from the Activity).

  - If the Activity does not have an outgoing Sequence Flow, then the Activity marks the end of one or more paths in the Process. When the Activity ends and there are no other parallel paths active, then the Process MUST be completed.

  - There are two exceptions to this: Compensation Activities and Event Sub-Processes.
Message Flow Connections

See “Message Flow Connection Rules” on page 43 for the entire set of objects and how they MAY be sources or targets of Message Flows.

Note – All Message Flows MUST connect two separate Pools. They MAY connect to the Pool boundary or to Flow Objects within the Pool boundary. They MUST NOT connect two objects within the same Pool.

◆ An Activity MAY be the target of a Message Flow; it can have zero (0) or more incoming Message Flows.
◆ An Activity MAY be a source of a Message Flow; it can have zero (0) or more outgoing Message Flows.

10.2.1 Resource Assignment

The following sections define how Resources can be defined for an Activity. Figure 10.7 displays the class diagram for the BPMN elements used for Resource assignment.

Figure 10.7 - The class diagram for assigning Resources

Resource Role

The ResourceRole element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 10.5 presents the additional model associations of the ResourceRole element.
Expression Assignment

Resources can be assigned to an Activity using Expressions. These Expressions MUST return Resource entity related data types, like Users or Groups. Different Expressions can return multiple Resources. All of them are assigned to the respective subclass of the ResourceRole element, for example as potential owners. The semantics is defined by the subclass.

The ResourceAssignmentExpression element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 10.6 presents the additional model associations of the ResourceAssignmentExpression element.

Parameterized Resource Assignment

Resources support query parameters that are passed to the Resource query at runtime. Parameters MAY refer to Task instance data using Expressions. During Resource query execution, an infrastructure can decide which of the Parameters defined by the Resource are used. It MAY use zero (0) or more of the Parameters specified. It MAY also override certain Parameters with values defined during Resource deployment. The deployment mechanism for Tasks and Resources is out of scope for this specification. Resource queries are evaluated to determine the set of Resources, e.g., people, assigned to the Activity. Failed Resource queries are treated like Resource queries that return an empty result set. Resource queries return one Resource or a set of Resources.

The ResourceParameterBinding element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 10.7 presents the additional model associations of the ResourceParameterBinding element.
### 10.2.2 Performer

The Performer class defines the resource that will perform or will be responsible for an Activity. The performer can be specified in the form of a specific individual, a group, an organization role or position, or an organization.

The Performer element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to ResourceRole, but does not have any additional attributes or model associations.

### 10.2.3 Tasks

A Task is an atomic Activity within a Process flow. A Task is used when the work in the Process cannot be broken down to a finer level of detail. Generally, an end-user and/or applications are used to perform the Task when it is executed.

A Task object shares the same shape as the Sub-Process, which is a rectangle that has rounded corners (see Figure 10.8).

- A Task is a rounded corner rectangle that MUST be drawn with a single thin line.
- The use of text, color, size, and lines for a Task MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 41.
- A boundary drawn with a thick line SHALL be reserved for Call Activity (Global Tasks) (see page 187).
- A boundary drawn with a dotted line SHALL be reserved for Event Sub-Processes (see page 176) and thus are not allowed for Tasks.
- A boundary drawn with a double line SHALL be reserved for Transaction Sub-Processes (see page 178) and thus are not allowed for Tasks.

---

#### Table 10.7  – ResourceParameterBinding model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameterRef: ResourceParameter</td>
<td>Reference to the parameter defined by the Resource.</td>
</tr>
<tr>
<td>expression: Expression</td>
<td>The Expression that evaluates the value used to bind the ResourceParameter.</td>
</tr>
</tbody>
</table>

---

BPMN specifies three types of markers for Task: a Loop marker or a Multi-Instance marker and a Compensation marker. A Task MAY have one or two of these markers (see Figure 10.9).
◆ The marker for a Task that is a standard loop MUST be a small line with an arrowhead that curls back upon itself. See page 189 for more information on loop Activities.

◆ The loop Marker MAY be used in combination with the compensation marker.

◆ The marker for a Task that is a multi-instance MUST be a set of three vertical lines. See page 191 for more information on multi-instance Activities.

◆ If the multi-instance instances are set to be performed in sequence rather than parallel, then the marker will be rotated 90 degrees (see Figure 10.49).

◆ The multi-instance marker MAY be used in combination with the compensation marker.

◆ The marker for a Task that is used for compensation MUST be a pair of left facing triangles (like a tape player “rewind” button). See page 302 for more information on compensation.

◆ The Compensation Marker MAY be used in combination with the loop marker or the multi-instance marker.

All the markers that are present MUST be grouped and the whole group centered at the bottom of the shape.

Figure 10.9 - Task markers

Figure 10.10 displays the class diagram for the Task element.

Figure 10.10 - The Task class diagram

The Task inherits the attributes and model associations of Activity (see Table 10.3). There are no further attributes or model associations of the Task.
10.2.3.1 Types of Tasks

There are different types of Tasks identified within BPMN to separate the types of inherent behavior that Tasks might represent. The list of Task types MAY be extended along with any corresponding indicators. A Task which is not further specified is called Abstract Task (this was referred to as the None Task in BPMN 1.2). The notation of the Abstract Task is shown in Figure 10.8.

Service Task

A Service Task is a Task that uses some sort of service, which could be a Web service or an automated application.

A Service Task object shares the same shape as the Task, which is a rectangle that has rounded corners. However, there is a graphical marker in the upper left corner of the shape that indicates that the Task is a Service Task (see Figure 10.11).

A Service Task is a rounded corner rectangle that MUST be drawn with a single thin line and includes a marker that distinguishes the shape from other Task types (as shown in Figure 10.11).

![Figure 10.11 - A Service Task Object](image)

The Service Task inherits the attributes and model associations of Activity (see Table 10.3). In addition the following constraints are introduced when the Service Task references an Operation: The Service Task has exactly one inputSet and at most one outputSet. It has a single Data Input with an ItemDefinition equivalent to the one defined by the Message referenced by the inMessageRef attribute of the associated Operation. If the Operation defines output Messages, the Service Task has a single Data Output that has an ItemDefinition equivalent to the one defined by the Message referenced by the outMessageRef attribute of the associated Operation.

The actual Participant whose service is used can be identified by connecting the Service Task to a Participant using a Message Flows within the definitional Collaboration of the Process – see Table 10.1.
The **Service Task** inherits the attributes and model associations of **Activity** (see Table 10.3). Table 10.8 presents additional the model associations of the **Service Task**.

**Table 10.8 – Service Task model associations**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>implementation: string  = ##webService</td>
<td>This attribute specifies the technology that will be used to send and receive the <strong>Messages</strong>. Valid values are &quot;##unspecified&quot; for leaving the implementation technology open, &quot;##WebService&quot; for the Web service technology or a URI identifying any other technology or coordination protocol. A Web service is the default technology.</td>
</tr>
<tr>
<td>operationRef: Operation [0..1]</td>
<td>This attribute specifies the operation that is invoked by the Service Task.</td>
</tr>
</tbody>
</table>

**Send Task**

A **Send Task** is a simple **Task** that is designed to send a **Message** to an external **Participant** (relative to the **Process**). Once the **Message** has been sent, the **Task** is completed.

The actual **Participant** which the **Message** is sent can be identified by connecting the **Send Task** to a **Participant** using a **Message Flows** within the definitional **Collaboration** of the **Process** – see Table 10.1.

A **Send Task** object shares the same shape as the **Task**, which is a rectangle that has rounded corners. However, there is a filled envelope marker (the same marker as a throw **Message Event**) in the upper left corner of the shape that indicates that the **Task** is a **Send Task**.
A **Send Task** is a rounded corner rectangle that MUST be drawn with a single thin line and includes a filled envelope marker that distinguishes the shape from other **Task** types (as shown in Figure 10.13).

![Send Task Object](image)

**Figure 10.13 - A Send Task Object**

```
Activity
   - isForCompensation
   - startQuantity
   - completionQuantity

Task
   - isActivity

SendTask
   - implementation
   - operationRef
   - inputSet
   - Data Input
     - itemDefinition

ReceiveTask
   - implementation
   - instantiate

Message
   - name
   - outMessageRef

Operation
   - name
   - operationRef
```

**Figure 10.14 - The Send Task and Receive Task class diagram**

The **Send Task** inherits the attributes and model associations of **Activity** (see Table 10.3). In addition the following constraints apply when the **Send Task** references a **Message**: The **Send Task** has at most one **inputSet** and one **Data Input**. If the **Data Input** is present, it MUST have an **ItemDefinition** equivalent to the one defined by the associated **Message**. At execution time, when the **Send Task** is executed, the data automatically moves from the **Data Input** on the **Send Task** into the **Message** to be sent. If the **Data Input** is not present, the **Message** will not be populated with data from the **Process**.

Table 10.9 presents the additional model associations of the **Send Task**.
### Receive Task

A **Receive Task** is a simple **Task** that is designed to wait for a **Message** to arrive from an external **Participant** (relative to the **Process**). Once the **Message** has been received, the **Task** is completed.

The actual **Participant** from which the **Message** is received can be identified by connecting the **Receive Task** to a **Participant** using a **Message Flows** within the definitional **Collaboration** of the **Process** – see Table 10.1.

A **Receive Task** is often used to start a **Process**. In a sense, the **Process** is bootstrapped by the receipt of the **Message**. In order for the **Receive Task** to instantiate the **Process** its **instantiate** attribute MUST be set to **true** and it MUST NOT have any incoming **Sequence Flow**.

A **Receive Task** object shares the same shape as the **Task**, which is a rectangle that has rounded corners. However, there is an unfilled envelope marker (the same marker as a **catch Message Event**) in the upper left corner of the shape that indicates that the **Task** is a **Receive Task**.

A **Receive Task** is a rounded corner rectangle that MUST be drawn with a single thin line and includes an unfilled envelope marker that distinguishes the shape from other **Task** types (as shown in Figure 10.15). If the **instantiate** attribute is set to **true**, the envelope marker looks like a **Message Start Event** (as shown in Figure 10.16).

---

**Table 10.9 – Send Task model associations**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>messageRef: Message [0..1]</td>
<td>A <strong>Message</strong> for the <strong>messageRef</strong> attribute MAY be entered. This indicates that the <strong>Message</strong> will be sent by the <strong>Task</strong>. The <strong>Message</strong> in this context is equivalent to an out-only message pattern (Web service). One or more corresponding outgoing <strong>Message Flows</strong> MAY be shown on the diagram. However, the display of the <strong>Message Flows</strong> is NOT REQUIRED. The <strong>Message</strong> is applied to all outgoing <strong>Message Flows</strong> and the <strong>Message</strong> will be sent down all outgoing <strong>Message Flows</strong> at the completion of a single instance of the <strong>Task</strong>.</td>
</tr>
<tr>
<td>operationRef: Operation</td>
<td>This attribute specifies the operation that is invoked by the <strong>Send Task</strong>.</td>
</tr>
<tr>
<td>implementation: string = ##webService</td>
<td>This attribute specifies the technology that will be used to send and receive the <strong>Messages</strong>. Valid values are &quot;##unspecified&quot; for leaving the implementation technology open, &quot;##WebService&quot; for the Web service technology or a URI identifying any other technology or coordination protocol. A Web service is the default technology.</td>
</tr>
</tbody>
</table>
The **Receive Task** inherits the attributes and model associations of **Activity** (see Table 10.3). In addition the following constraints apply when the **Receive Task** references a **Message**: The **Receive Task** has at most one **outputSet** and at most one **Data output**. If the **Data output** is present, it MUST have an **ItemDefinition** equivalent to the one defined by the associated **Message**. At execution time, when the **Receive Task** is executed, the data automatically moves from the **Message** to the **Data Output** on the **Receive Task**. If the **Data Output** is not present, the payload within the **Message** will not flow out of the **Receive Task** and into the **Process**.

Table 10.10 presents the additional attributes and model associations of the **Receive Task**.

### Table 10.10 - Receive Task attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>messageRef</strong></td>
<td>A <strong>Message</strong> for the <strong>messageRef</strong> attribute MAY be entered. This indicates that the <strong>Message</strong> will be received by the <strong>Task</strong>. The <strong>Message</strong> in this context is equivalent to an <em>in-only</em> message pattern (Web service). One (1) or more corresponding <em>incoming Message Flows</em> MAY be shown on the diagram. However, the display of the <strong>Message Flows</strong> is NOT REQUIRED. The <strong>Message</strong> is applied to all <em>incoming Message Flows</em>, but can arrive for only one (1) of the <em>incoming Message Flows</em> for a single <em>instance</em> of the <strong>Task</strong>.</td>
</tr>
<tr>
<td><strong>instantiate</strong></td>
<td><strong>Receive Tasks</strong> can be defined as the instantiation mechanism for the <strong>Process</strong> with the <strong>instantiate</strong> attribute. This attribute MAY be set to <em>true</em> if the <strong>Task</strong> is the first <strong>Activity</strong> (i.e., there are no <em>incoming Sequence Flows</em>). Multiple <strong>Tasks</strong> MAY have this attribute set to <em>true</em>.</td>
</tr>
<tr>
<td><strong>operationRef</strong></td>
<td>This attribute specifies the operation through which the <strong>Receive Task</strong> receives the <strong>Message</strong>.</td>
</tr>
<tr>
<td><strong>implementation</strong></td>
<td>This attribute specifies the technology that will be used to send and receive the <strong>Messages</strong>. Valid values are &quot;##unspecified&quot; for leaving the implementation technology open, &quot;##WebService&quot; for the Web service technology or a URI identifying any other technology or coordination protocol A Web service is the default technology.</td>
</tr>
</tbody>
</table>
User Task

A **User Task** is a typical “workflow” **Task** where a human performer performs the **Task** with the assistance of a software application and is scheduled through a task list manager of some sort.

A **User Task** is a rounded corner rectangle that MUST be drawn with a single thin line and includes a human figure marker that distinguishes the shape from other **Task** types (as shown in Figure 10.17).

![Figure 10.17 - A User Task Object](image)

See “User Task” on page 163 within the larger section of “Human Interactions” for the details of **User Tasks**.

Manual Task

A **Manual Task** is a **Task** that is expected to be performed without the aid of any business process execution engine or any application. An example of this could be a telephone technician installing a telephone at a customer location.

A **Manual Task** is a rounded corner rectangle that MUST be drawn with a single thin line and includes a hand figure marker that distinguishes the shape from other **Task** types (as shown in Figure 10.17).

![Figure 10.18 - A Manual Task Object](image)

See “Manual Task” on page 165 within the larger section of “Human Interactions” for the details of **Manual Tasks**.

Business Rule

A **Business Rule Task** provides a mechanism for the **Process** to provide input to a Business Rules Engine and to get the output of calculations that the Business Rules Engine might provide. The **InputOutputSpecification** of the **Task** (see page 211) will allow the **Process** to send data to and receive data from the Business Rules Engine.

A **Business Rule Task** object shares the same shape as the **Task**, which is a rectangle that has rounded corners. However, there is a graphical marker in the upper left corner of the shape that indicates that the **Task** is a **Business Rule Task** (see Figure 10.11).
A **Business Rule Task** is a rounded corner rectangle that MUST be drawn with a single thin line and includes a marker that distinguishes the shape from other **Task** types (as shown in Figure 10.19).

![Figure 10.19 - A Business Rule Task Object](image)

The **Business Rule Task** inherits the attributes and model associations of **Activity** (see Table 10.3). Table 10.11 presents the additional attributes of the **Business Rule Task**.

<table>
<thead>
<tr>
<th>Attribute Name: implementation: string = ##unspecified</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>This attribute specifies the technology that will be used to implement the <strong>Business Rule Task</strong>. Valid values are &quot;##unspecified&quot; for leaving the implementation technology open, &quot;##WebService&quot; for the Web service technology or a URI identifying any other technology or coordination protocol. The default technology for this task is unspecified.</td>
<td></td>
</tr>
</tbody>
</table>

**Script Task**

A **Script Task** is executed by a business process engine. The modeler or implementer defines a script in a language that the engine can interpret. When the **Task** is ready to start, the engine will execute the script. When the script is completed, the **Task** will also be completed.

A **Script Task** object shares the same shape as the **Task**, which is a rectangle that has rounded corners. However, there is a graphical marker in the upper left corner of the shape that indicates that the **Task** is a **Script Task** (see Figure 10.11).

A **Script Task** is a rounded corner rectangle that MUST be drawn with a single thin line and includes a marker that distinguishes the shape from other **Task** types (as shown in Figure 10.20).

![Figure 10.20 - A Script Task Object](image)
The **Script Task** inherits the attributes and model associations of **Activity** (see Table 10.3). Table 10.12 presents the additional attributes of the **Script Task**.

### Table 10.12 – Script Task attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scriptFormat</code></td>
<td>Defines the format of the script. This attribute value MUST be specified with a mime-type format. And it MUST be specified if a script is provided.</td>
</tr>
<tr>
<td><code>script</code></td>
<td>The modeler MAY include a script that can be run when the Task is performed. If a script is not included, then the Task will act as the equivalent of an Abstract Task.</td>
</tr>
</tbody>
</table>

#### 10.2.4 Human Interactions

**10.2.4.1 Tasks with Human involvement**

In many business workflows, human involvement is needed to complete certain **Tasks** specified in the workflow model. **BPMN** specifies two different types of **Tasks** with human involvement, the **Manual Task** and the **User Task**.

A **User Task** is executed by and managed by a business process runtime. Attributes concerning the human involvement, like people assignments and UI rendering can be specified in great detail. A **Manual Task** is neither executed by nor managed by a business process runtime.

**Notation**

Both, the **Manual Task** and **User Task** share the same shape, which is a rectangle that has rounded corners. **Manual Tasks** and **User Tasks** have a Icons to indicate the human involvement is REQUIRED to complete the Task (see Figure 10.15 and Figure 10.17).

**Manual Task**

A **Manual Task** is a **Task** that is not managed by any business process engine. It can be considered as an unmanaged **Task**, unmanaged in the sense of that the business process engine doesn’t track the start and completion of such a **Task**. An example of this could be a paper based instruction for a telephone technician to install a telephone at a customer location.
The **User Task** inherits the attributes and model associations of **Activity** (see Table 10.3), but does not have any additional attributes or model associations.

**User Task**

A **User Task** is a typical “workflow” **Task** where a human performer performs the **Task** with the assistance of a software application. The lifecycle of the **Task** is managed by a software component (called task manager) and is typically executed in the context of a **Process**.

The **User Task** can be implemented using different technologies, specified by the implementation attribute. Besides the Web service technology, any technology can be used. A **User Task** for instance can be implemented using WS-HumanTask by setting the implementation attribute to "http://docs.oasis-open.org/ns/bpel4people/ws-humantask/protocol/200803."
The **User Task** inherits the attributes and model associations of **Activity** (see Table 10.3). Table 10.13 presents the additional attributes and model associations of the **User Task**. If implementations extend these attributes (e.g., to introduce subjects or descriptions with presentation parameters), they SHOULD use attributes defined by the OASIS WS-HumanTask specification.

**Table 10.13 - User Task attributes and model associations**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>implementation: string = ##unspecified</td>
<td>This attribute specifies the technology that will be used to implement the <strong>User Task</strong>. Valid values are &quot;##unspecified&quot; for leaving the implementation technology open, &quot;##WebService&quot; for the Web service technology or a URI identifying any other technology or coordination protocol. The default technology for this task is unspecified.</td>
</tr>
<tr>
<td>renderings: Rendering [0..*]</td>
<td>This attributes acts as a hook which allows BPMN adopters to specify task rendering attributes by using the BPMN Extension mechanism.</td>
</tr>
</tbody>
</table>

The **User Task** inherits the instance attributes of **Activity** (see Table 8.49). Table 10.14 presents the instance attributes of the **User Task** element.

**Table 10.14 - User Task instance attributes**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>actualOwner: string</td>
<td>Returns the “user” who picked/claimed the User task and became the actual owner of it. The value is a literal representing the user’s id, email address etc.</td>
</tr>
<tr>
<td>taskPriority: integer</td>
<td>Returns the priority of the <strong>User Task</strong>.</td>
</tr>
</tbody>
</table>

**Rendering of User Tasks**

**BPMN User Tasks** need to be rendered on user interfaces like forms clients, portlets, etc. The Rendering element provides an extensible mechanism for specifying UI renderings for **User Tasks** (Task UI). The element is optional. One or more rendering methods can be provided in a Task definition. A **User Task** can be deployed on any compliant implementation, irrespective of the fact whether the implementation supports specified rendering methods or not. The Rendering element is the extension point for renderings. Things like language considerations are opaque for the Rendering element because the rendering applications typically provide Multilanguage support. Where this is not the case, providers of certain rendering types can decide to extend the rendering type in order to provide language information for a given rendering. The content of the rendering element is not defined by this specification.

**Human Performers**

People can be assigned to **Activities** in various roles (called “generic human roles” in WS-HumanTask). **BPMN 1.2** traditionally only has the **Performer** role. In addition to supporting the **Performer** role, **BPMN 2.0** defines a specific **HumanPerformer** element allowing specifying more specific human roles as specialization of **HumanPerformer**, such as **PotentialOwner**.
Figure 10.23 - HumanPerformer class diagram

The HumanPerformer element inherits the attributes and model associations of ResourceRole (see Table 10.5), through its relationship to Performer, but does not have any additional attributes or model associations.

Potential Owners

Potential owners of a User Task are persons who can claim and work on it. A potential owner becomes the actual owner of a Task, usually by explicitly claiming it.

XML Schema for Human Interactions

Table 10.15 – ManualTask XML schema

```xml
<xsd:element name="manualTask" type="tManualTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tManualTask">
  <xsd:complexContent>
    <xsd:extension base="tTask">
      <xsd:complexType>
        <xsd:complexContent>
          <xsd:extension base="tTask"/>
        </xsd:complexType>
      </xsd:complexType>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
Table 10.16 – UserTask XML schema

```xml
<xsd:element name="userTask" type="tUserTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tUserTask">
    <xsd:complexContent>
        <xsd:extension base="tTask">
            <xsd:sequence>
                <xsd:element ref="rendering" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="implementation" type="tImplementation"
                default="##unspecified"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:element name="rendering" type="tRendering"/>
<xsd:complexType name="tRendering">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement"/>
    </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="tImplementation">
    <xsd:union memberTypes="xsd:anyURI">
        <xsd:simpleType>
            <xsd:restriction base="xsd:token">
                <xsd:enumeration value="##unspecified"/>
                <xsd:enumeration value="##WebService"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:union>
</xsd:simpleType>
```

Table 10.17 – HumanPerformer XML schema

```xml
<xsd:element name="humanPerformer" type="tHumanPerformer" substitutionGroup="performer"/>
<xsd:complexType name="tHumanPerformer">
    <xsd:complexContent>
        <xsd:extension base="tPerformer">
            <xsd:sequence>
                <xsd:element ref="peopleAssignment" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```
Consider the following sample procurement Process from the Buyer perspective (see Figure 10.24).

The Process comprises of two User Tasks

- **Approve Order**: After the quotation handling, the order needs to be approved by some regional manager to continue with the order and shipment handling.
- **Review Order**: Once the order has been shipped to the Buyer, the order and shipment documents will be reviewed again by someone.

The details of the Resource and resource assignments are not shown in the BPMN above. See below XML sample of the “Buyer” Process for the Resource usage and resource assignments for potential owners.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions id="Definition"

targetNamespace="http://www.example.org/UserTaskExample"

typeLanguage="http://www.w3.org/2001/XMLSchema"

expressionLanguage="http://www.w3.org/1999/XPath"

xmlns="http://www.w3.org/2001/XMLSchema"

xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
```
xmlns:tns="http://www.example.org/UserTaskExample">

<resource id="regionalManager" name="Regional Manager">
    <resourceParameter id="buyerName" isRequired="true" name="Buyer Name" type="xsd:string"/>
    <resourceParameter id="region" isRequired="false" name="Region" type="xsd:string"/>
</resource>

<resource id="departmentalReviewer" name="Departmental Reviewer">
    <resourceParameter id="buyerName" isRequired="true" name="Buyer Name" type="xsd:string"/>
</resource>

<collaboration id="BuyerCollaboration" name="Buyer Collaboration">
    <participant id="BuyerParticipant" name="Buyer" processRef="BuyerProcess"/>
</collaboration>

<!-- Process definition -->
<process id="BuyerProcess" name="Buyer Process">

<laneSet id="BuyerLaneSet">
    <lane id="BuyerLane">
        <flowNodeRef>StartProcess</flowNodeRef>
        <flowNodeRef>QuotationHandling</flowNodeRef>
        <flowNodeRef>ApproveOrder</flowNodeRef>
        <flowNodeRef>OrderApprovedDecision</flowNodeRef>
        <flowNodeRef>TerminateProcess</flowNodeRef>
        <flowNodeRef>OrderAndShipment</flowNodeRef>
        <flowNodeRef>OrderHandling</flowNodeRef>
        <flowNodeRef>ShipmentHandling</flowNodeRef>
        <flowNodeRef>OrderAndShipmentMerge</flowNodeRef>
        <flowNodeRef>ReviewOrder</flowNodeRef>
        <flowNodeRef>EndProcess</flowNodeRef>
    </lane>
</laneSet>

<startEvent id="StartProcess"/>

<sequenceFlow sourceRef="StartProcess" targetRef="QuotationHandling"/>

<task id="QuotationHandling" name="Quotation Handling"/>

<sequenceFlow sourceRef="QuotationHandling" targetRef="ApproveOrder"/>

<userTask id="ApproveOrder" name="ApproveOrder">
    <potentialOwner>
        <resourceRef>tns:regionalManager</resourceRef>
        <resourceParameterBinding parameterRef="tns:buyerName"/>
    </potentialOwner>
</userTask>
10.2.5 Sub-Processes

A **Sub-Process** is an **Activity** whose internal details have been modeled using **Activities**, **Gateways**, **Events**, and **Sequence Flows**. A **Sub-Process** is a graphical object within a **Process**, but it also can be “opened up” to show a lower-level **Process**. **Sub-Processes** define a contextual **scope** that can be used for attribute visibility, transactional **scope**, for the handling of **exceptions** (see page 275 for more details), of **Events**, or for **compensation** (see page 302 for more details).

There are different types of **Sub-Processes**, which will be described in the next five sections.

**Embedded Sub-Process (Sub-Process)**

A **Sub-Process** object shares the same shape as the **Task** object, which is a rounded rectangle.

- A **Sub-Process** is a rounded corner rectangle that MUST be drawn with a single thin line.
- The use of text, color, size, and lines for a **Sub-Process** MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 41 with the exception that:
  - A boundary drawn with a thick line SHALL be reserved for **Call Activity (Sub-Processes)** (see page 183).
  - A boundary drawn with a dotted line SHALL be reserved for **Event Sub-Processes** (see page 176).
  - A boundary drawn with a double line SHALL be reserved for **Transaction Sub-Processes** (see page 178).

The **Sub-Process** can be in a collapsed view that hides its details (see Figure 10.25) or a **Sub-Process** can be in an expanded view that shows its details within the view of the **Process** in which it is contained (see Figure 10.26). In the collapsed form, the **Sub-Process** object uses a marker to distinguish it as a **Sub-Process**, rather than a **Task**.

- The **Sub-Process** marker MUST be a small square with a plus sign (+) inside. The square MUST be positioned at the bottom center of the shape.
They are used to create a context for exception handling that applies to a group of **Activities** (see page 275 for more details). **Compensations** can be handled similarly (see page 302 for more details).

Expanded **Sub-Processes** can be used as a mechanism for showing a group of parallel **Activities** in a less-cluttered, more compact way. In Figure 10.27, **Activities** “C” and “D” are enclosed in an unlabeled expanded **Sub-Process**. These two **Activities** will be performed in parallel. Notice that the expanded **Sub-Process** does not include a **Start Event** or an **End Event** and the **Sequence Flows** to/from these **Events**. This usage of expanded **Sub-Processes** for “parallel boxes” is the motivation for having **Start** and **End Events** being optional objects.
BPMN specifies five types of standard markers for Sub-Processes. The (Collapsed) Sub-Process marker, seen in Figure 10.24, can be combined with four other markers: a loop marker or a multi-instance marker, a Compensation marker, and an Ad-Hoc marker. A collapsed Sub-Process MAY have one to three of these other markers, in all combinations except that loop and multi-instance cannot be shown at the same time (see Figure 10.28).

- The marker for a Sub-Process that loops MUST be a small line with an arrowhead that curls back upon itself.
- The loop marker MAY be used in combination with any of the other markers except the multi-instance marker.
- The marker for a Sub-Process that has multiple instances MUST be a set of three vertical lines in parallel.
- The multi-instance marker MAY be used in combination with any of the other markers except the loop marker.
- The marker for an ad-hoc Sub-Process MUST be a “tilde” symbol.
- The ad-hoc marker MAY be used in combination with any of the other markers.
- The marker for a Sub-Process that is used for compensation MUST be a pair of left facing triangles (like a tape player “rewind” button).
- The Compensation marker MAY be used in combination with any of the other markers.
- All the markers that are present MUST be grouped and the whole group centered at the bottom of the Sub-Process.

Figure 10.28 - Collapsed Sub-Process Markers

The Sub-Process now corresponds to the Embedded Sub-Process of BPMN 1.2. The Reusable Sub-Process of BPMN 1.2 corresponds to the Call Activity (calling a Process - see page 183).

Figure 10.29 shows the class diagram related to Sub-Processes.
Figure 10.29- The Sub-Process class diagram

The Sub-Process element inherits the attributes and model associations of Activity (see Table 10.3) and of FlowElementContainer (see Table 8.45). Table 10.3 presents the additional attributes of the Sub-Process element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
</table>
| triggeredByEvent: boolean = false | A flag that identifies whether this Sub-Process is an Event Sub-Process.  
  • If false, then this Sub-Process is a normal Sub-Process.  
  • If true, then this Sub-Process is an Event Sub-Process and is subject to additional constraints (see page 176). |
| artifacts: Artifact [0..*] | This attribute provides the list of Artifacts that are contained within the Sub-Process. |

Reusable Sub-Process (Call Activity)

The reusable Sub-Process of BPMN 1.2 corresponds to the Call Activity that calls a pre-defined Process. See details of a Call Activity on page 183.

Event Sub-Process

An Event Sub-Process is a specialized Sub-Process that is used within a Process (or Sub-Process). A Sub-Process is defined as an Event Sub-Process when its triggeredByEvent attribute is set to true.

An Event Sub-Process is not part of the normal flow of its parent Process—there are no incoming or outgoing Sequence Flows.
◆ An Event Sub-Process MUST NOT have any incoming or outgoing Sequence Flows.

An Event Sub-Process MAY or MAY NOT occur while the parent Process is active, but it is possible that it will occur many times. Unlike a standard Sub-Process, which uses the flow of the parent Process as a trigger, an Event Sub-Process has a Start Event with a trigger. Each time the Start Event is triggered while the parent Process is active, then the Event Sub-Process will start.

◆ The Start Event of an Event Sub-Process MUST have a defined trigger.

◆ The Start Event trigger (EventDefinition) MUST be from the following types: Message, Error, Escalation, Compensation, Conditional, Signal, and Multiple (see page 260 for more details).

◆ An Event Sub-Process MUST have one and only one Start Event.

An Event Sub-Process object shares the same basic shape as the Sub-Process object, which is a rounded rectangle.

◆ An Event Sub-Process is a rounded corner rectangle that MUST be drawn with a single thin dotted line (see Figure 10.30 and Figure 10.31).

◆ The use of text, color, size, and lines for an Event Sub-Process MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 41 with the exception that:

◆ If the Event Sub-Process is collapsed, then its Start Event will be used as a marker in the upper left corner of the shape (see Figure 10.30).

![Figure 10.30 - An Event Sub-Process object (Collapsed)](image)

![Figure 10.31 - An Event Sub-Process object (expanded)](image)

There are two possible consequences to the parent Process when an Event Sub-Process is triggered: 1) the parent Process can be interrupted, and 2) the parent Process can continue its work (not interrupted). This is determined by the type of Start Event that is used. See page 242 for the list of interrupting and non-interrupting Event Sub-Process Start Events.
Figure 10.32 provides an example of a **Sub-Process** that includes three **Event Sub-Processes**. The first **Event Sub-Process** is triggered by a **Message**, does not interrupt the **Sub-Process**, and can occur multiple times. The second **Event Sub-Process** is used for **compensation** and will only occur after the **Sub-Process** has completed. The third **Event Sub-Process** handles errors that occur while the **Sub-Process** is active and will stop (interrupt) the **Sub-Process** if triggered.

![Diagram of Event Sub-Processes](image)

**Figure 10.32 - An example that includes Event Sub-Processes**

**Transaction**

A **Transaction** is a specialized type of **Sub-Process** that will have a special behavior that is controlled through a transaction protocol (such as WS-Transaction). The boundary of the **Sub-Process** will be double-lined to indicate that it is a **Transaction** (see Figure 10.33).
A **Transaction Sub-Process** is a rounded corner rectangle that MUST be drawn with a double thin line.

The use of text, color, size, and lines for a transaction **Sub-Process** MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 41.

![Diagram of a Transaction Sub-Process](image)

**Figure 10.33 - A Transaction Sub-Process**

![Diagram of a Collapsed Transaction Sub-Process](image)

**Figure 10.34 - A Collapsed Transaction Sub-Process**

The **Transaction Sub-Process** element inherits the attributes and model associations of **Activities** (see Table 10.3) through its relationship to **Sub-Process**. Table 10.21 presents the additional attributes and model associations of the **Transaction Sub-Process**.
There are three basic outcomes of a Transaction:

1. **Successful completion**: this will be shown as a normal Sequence Flow that leaves the Transaction Sub-Process.

2. **Failed completion (Cancel)**: When a Transaction is cancelled, the Activities inside the Transaction will be subjected to the cancellation actions, which could include rolling back the Process and compensation (see page 302 for more information on compensation) for specific Activities. Note that other mechanisms for interrupting a Transaction Sub-Process will not cause compensation (e.g., Error, Timer, and anything for a non-Transaction Activity). A Cancel Intermediate Event, attached to the boundary of the Activity, will direct the flow after the Transaction has been rolled back and all compensation has been completed. The Cancel Intermediate Event can only be used when attached to the boundary of a Transaction Sub-Process. It cannot be used in any normal flow and cannot be attached to a non-Transaction Sub-Process. There are two mechanisms that can signal the cancellation of a Transaction:
   - A Cancel End Event is reached within the transaction Sub-Process. A Cancel End Event can only be used within a transaction Sub-Process.
   - A cancel Message can be received via the transaction protocol that is supporting the execution of the Transaction Sub-Process.

3. **Hazard**: This means that something went terribly wrong and that a normal success or cancel is not possible. Error Intermediate Events are used to show Hazards. When a Hazard happens, the Activity is interrupted (without compensation) and the flow will continue from the Error Intermediate Event.

The behavior at the end of a successful Transaction Sub-Process is slightly different than that of a normal Sub-Process. When each path of the Transaction Sub-Process reaches a non-Cancel End Event(s), the flow does not immediately move back up to the higher-level parent Process, as does a normal Sub-Process. First, the transaction protocol needs to verify that all the Participants have successfully completed their end of the Transaction. Most of the time this will be true and the flow will then move up to the higher-level Process. But it is possible that one of the Participants can end up with a problem that causes a Cancel or a Hazard. In this case, the flow will then move to the appropriate Intermediate Event, even though it had apparently finished successfully.

### Ad-Hoc Sub-Process

An Ad-Hoc Sub-Process is a specialized type of Sub-Process that is a group of Activities that have no REQUIRED sequence relationships. A set of Activities can be defined for the Process, but the sequence and number of performances for the Activities is determined by the performers of the Activities.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>method: Transaction-Method</td>
<td>The method is an attribute that defines the Transaction method used to commit or cancel a Transaction. For executable Processes, it SHOULD be set to a technology specific URI, e.g., <a href="http://schemas.xmlsoap.org/ws/2004/10/wsat">http://schemas.xmlsoap.org/ws/2004/10/wsat</a> for WS-AtomicTransaction, or <a href="http://docs.oasis-open.org/ws-tx/wsba/2006/06/AtomicOutcome">http://docs.oasis-open.org/ws-tx/wsba/2006/06/AtomicOutcome</a> for WS-BusinessActivity. For compatibility with BPMN 1.1, it can also be set to &quot;##compensate&quot;, &quot;##store&quot;, or &quot;##image&quot;.</td>
</tr>
</tbody>
</table>
A Sub-Process is marked as being ad-hoc with a “tilde” symbol placed at the bottom center of the Sub-Process shape (see Figure 10.35 and Figure 10.36).

- The marker for an Ad-Hoc Sub-Process MUST be a “tilde” symbol.
- The Ad-Hoc Marker MAY be used in combination with any of the other markers.

![Figure 10.35 - A collapsed Ad-Hoc Sub-Process](image)

![Figure 10.36 - An expanded Ad-Hoc Sub-Process](image)

The Ad-Hoc Sub-Process element inherits the attributes and model associations of Activities (see Table 10.3) through its relationship to Sub-Process. Table 10.22 presents the additional model associations of the Ad-Hoc Sub-Process.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>completionCondition: expression</td>
<td>This Expression defines the conditions when the Process will end. When the Expression is evaluated to true, the Process will be terminated.</td>
</tr>
<tr>
<td>ordering: AdHocOrdering = Parallel { Parallel</td>
<td>Sequential }</td>
</tr>
<tr>
<td>cancelRemaining-Instances: boolean = true</td>
<td>This attribute is used only if ordering is parallel. It determines whether running instances are cancelled when the completionCondition becomes true.</td>
</tr>
</tbody>
</table>
Activities within the Process are generally disconnected from each other. During execution of the Process, any one or more of the Activities MAY be active and they MAY be performed multiple times. The performers determine when Activities will start, what the next Activity will be, and so on.

Examples of the types of Processes that are Ad-Hoc include computer code development (at a low level), sales support, and writing a book chapter. If we look at the details of writing a book chapter, we could see that the Activities within this Process include: researching the topic, writing text, editing text, generating graphics, including graphics in the text, organizing references, etc. (see Figure 10.37). There MAY be some dependencies between Tasks in this Process, such as writing text before editing text, but there is not necessarily any correlation between an instance of writing text to an instance of editing text. Editing can occur infrequently and based on the text of many instances of the writing text Task.

![Figure 10.37 - An Ad-Hoc Sub-Process for writing a book chapter](image)

Although there is no explicit Process structure, some sequence and data dependencies can be added to the details of the Process. For example, we can extend the book chapter Ad-Hoc Sub-Process shown above and add Data Objects, Data Associations, and even Sequence Flows (Figure 10.38).

Ad-Hoc Sub-Processes restrict the use of BPMN elements that would normally be used in Sub-Processes.

- The list of BPMN elements that MUST be used in an Ad-Hoc Sub-Process: Activity.
- The list of BPMN elements that MAY be used in an Ad-Hoc Sub-Process: Data Object, Sequence Flow, Association, Data Association, Group, Message Flow (as a source or target), Gateway, and Intermediate Event.
- The list of BPMN elements that MUST NOT be used in an Ad-Hoc Sub-Process: Start Event, End Event, Conversations (graphically), Conversation Links (graphically), and Choreography Activities.
The **Data Objects** act as inputs into the **Tasks** act as an additional constraint for the performance of those **Tasks**. The **performers** still determine when the **Tasks** will be performed, but they are now constrained in that they cannot start the **Task** without the appropriate input. The addition of **Sequence Flows** between the **Tasks** (e.g., between “Generate Graphics” and “Include Graphics in Text”) creates a dependency where the performance of the first **Task** MUST be followed by a performance of the second **Task**. This does not mean that the second **Task** is to be performed immediately, but there MUST be a performance of the second **Task** after the performance of the first **Task**.

It is a challenge for a BPM engine to monitor the status of **Ad-Hoc Sub-Processes**, usually these kind of **Processes** are handled through groupware applications (such as e-mail), but **BPMN** allows modeling of **Processes** that are not necessarily executable, although there are some process engines that can follow an **Ad-Hoc Sub-Process**. Given this, at some point the **Ad-Hoc Sub-Process** will have complete and this can be determined by evaluating a **completionCondition** that evaluates **Process** attributes that will have been updated by an **Activity** in the **Process**.

### 10.2.6 Call Activity

A **Call Activity** identifies a point in the **Process** where a global **Process** or a **Global Task** is used. The **Call Activity** acts as a ‘wrapper’ for the invocation of a global **Process** or **Global Task** within the execution. The activation of a call **Activity** results in the transfer of control to the called global **Process** or **Global Task**.

The **BPMN 2.0 Call Activity** corresponds to the **Reusable Sub-Process** of **BPMN 1.2**. A **BPMN 2.0 Sub-Process** corresponds to the **Embedded Sub-Process** of **BPMN 1.2** (see the previous section).
A Call Activity object shares the same shape as the Task and Sub-Process, which is a rectangle that has rounded corners. However, the target of what the Activity calls will determine the details of its shape.

- If the Call Activity calls a Global Task, then the shape will be the same as a Task, but the boundary of the shape will MUST have a thick line (see Figure 10.39).
  - The Call Activity MUST display the marker of the type of Global Task (e.g., the Call Activity would display the User Task marker if calling a Global User Task).
- If the Call Activity calls a Process, then there are two options:
  - The details of the called Process can be hidden and the shape of the Call Activity will be the same as a collapsed Sub-Process, but the boundary of the shape MUST have a thick line (see Figure 10.40).
  - If the details of the called Process are available, then the shape of the Call Activity will be the same as an expanded Sub-Process, but the boundary of the shape MUST have a thick line (see Figure 10.41).

Figure 10.39- A Call Activity object calling a Global Task

Figure 10.40 - A Call Activity object calling a Process (Collapsed)

Figure 10.41 - A Call Activity object calling a Process (Expanded)
When a **Process** with a *definitional Collaboration*, calls a **Process** that also has a *definitional Collaboration*, the **Participants** of the two **Collaborations** can be matched to each other using **ParticipantAssociations** of the **Collaboration** of the calling **Process**.

A **Call Activity** MUST fulfill the data requirements, as well as return the data produced by the **CallableElement** being invoked (see Figure 10.41). This means that the elements contained in the **Call Activity**’s **InputOutputSpecification** MUST exactly match the elements contained in the referenced **CallableElement**. This includes **DataInputs**, **DataOutputs**, **InputSets**, and **OutputSets**.

---

**Figure 10.42** - The Call Activity class diagram

A **Call Activity** can override properties and attributes of the element being called, potentially changing the behavior of the called element based on the calling context. For example, when the **Call Activity** defines one or more **ResourceRole** elements, the elements defined by the **CallableElement** are ignored and the elements defined in the **Call Activity** are used instead. Also, **Events** that are propagated along the hierarchy (errors and escalations) are propagated from the called element to the **Call Activity** (and can be handled on its boundary).

The **Call Activity** inherits the attributes and model associations of **Activity** (see Table 10.3). Table 10.23 presents the additional model associations of the **Call Activity**.
Callable Element

CallableElement is the abstract super class of all Activities that have been defined outside of a Process or Choreography but which can be called (or reused), by a Call Activity, from within a Process or Choreography. It MAY reference Interfaces that define the service operations that it provides. The BPMN elements that can be called by Call Activities (i.e., are CallableElements) are: Process and GlobalTask (see Figure 10.43). CallableElements are RootElements, which can be imported and used in other Definitions. When CallableElements (e.g., Process) are defined, they are contained within Definitions.

Figure 10.43 - CallableElement class diagram

The CallableElement inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to RootElement. Table 10.24 presents the additional attributes and model associations of the CallableElement.
When a CallableElement is exposed as a Service, it has to define one or more InputOutputBinding elements. An InputOutputBinding element binds one Input and one Output of the InputOutputSpecification to an Operation of a Service Interface. Table 10.25 presents the additional model associations of the InputOutputBinding.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string [0..1]</td>
<td>The descriptive name of the element.</td>
</tr>
<tr>
<td>supportedInterfaceRefs: Interface [0..*]</td>
<td>The Interfaces describing the external behavior provided by this element.</td>
</tr>
<tr>
<td>ioBinding: InputOutputBinding [0..*]</td>
<td>The InputOutputBinding defines a combination of one InputSet and one OutputSet in order to bind this to an operation defined in an interface.</td>
</tr>
</tbody>
</table>

Table 10.25 – InputOutputBinding model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputDataRef: DataInput</td>
<td>A reference to one specific DataInput defined as part of the InputOutputSpecification of the Activity.</td>
</tr>
<tr>
<td>outputDataRef: DataOutput</td>
<td>A reference to one specific DataOutput defined as part of the InputOutputSpecification of the Activity.</td>
</tr>
<tr>
<td>operationRef: Operation</td>
<td>A reference to one specific Operation defined as part of the Interface of the Activity.</td>
</tr>
</tbody>
</table>

10.2.7 Global Task

A Global Task is a reusable, atomic Task definition that can be called from within any Process by a Call Activity.
Figure 10.44 - Global Tasks class diagram

The GlobalTask inherits the attributes and model associations of Callable Element (see Table 10.24). Table 10.26 presents the additional model associations of the GlobalTask.

Table 10.26 – Global Task model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>resources: ResourceRole [0..*]</td>
<td>Defines the resource that will perform or will be responsible for the GlobalTask. In the case where the Call Activity that references this GlobalTask defines its own resources, they will override the ones defined here.</td>
</tr>
</tbody>
</table>

Types of Global Task

There are different types of Tasks identified within BPMN to separate the types of inherent behavior that Tasks might represent. The types of Global Tasks are only a subset of standard Tasks types. Only GlobalUserTask, GlobalManualTask, GlobalScriptTask, and GlobalBusinessRuleTask are defined in BPMN. For the sake of efficiency in this specification, the list of Task types is presented once on page 156. The behavior, attributes, and model associations defined in that section also apply to the corresponding types of Global Tasks.
10.2.8 Loop Characteristics

**Activities** MAY be repeated sequentially, essentially behaving like a loop. The presence of **LoopCharacteristics** signifies that the **Activity** has looping behavior. **LoopCharacteristics** is an abstract class. Concrete subclasses define specific kinds of looping behavior.

The **LoopCharacteristics** inherits the attributes and model associations of **BaseElement** (see Table 8.5). There are no further attributes or model associations of the **LoopCharacteristics**.

However, each **Loop Activity instance** has attributes whose values MAY be referenced by **Expressions**. These values are only available when the **Loop Activity** is being executed.

Figure 10.45 displays the class diagram for an **Activity's loop** characteristics, including the details of both the standard loop and a multi-instance.

![LoopCharacteristics class diagram](image)

The **LoopCharacteristics** element inherits the attributes and model associations of **BaseElement** (see Table 8.5), but does not have any further attributes or model associations. However, a **Loop Activity** does have additional **instance** attributes as shown in Table 10.27.
The **StandardLoopCharacteristics** class defines looping behavior based on a boolean condition. The **Activity** will loop as long as the boolean condition is true. The condition is evaluated for every loop iteration, and MAY be evaluated at the beginning or at the end of the iteration. In addition, a numeric cap can be optionally specified. The number of iterations MAY NOT exceed this cap.

- The marker for a **Task** or a **Sub-Process** that is a standard loop MUST be a small line with an arrowhead that curls back upon itself (see Figure 10.46 and Figure 10.47).
- The **loop** Marker MAY be used in combination with the **Compensation** Marker.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loopCounter: integer</code></td>
<td>The <strong>LoopCounter</strong> attribute is used at runtime to count the number of loops and is automatically updated by the process engine.</td>
</tr>
</tbody>
</table>

### Table 10.27 – Loop Activity instance attributes

The **StandardLoopCharacteristics** element inherits the attributes and model associations of **BaseElement** (see Figure 8.5), through its relationship to **LoopCharacteristics**. Table 10.28 presents the additional attributes and model associations for the **StandardLoopCharacteristics** element.
Multi-Instance Characteristics

The MultiInstanceLoopCharacteristics class allows for creation of a desired number of Activity instances. The instances MAY execute in parallel or MAY be sequential. Either an Expression is used to specify or calculate the desired number of instances or a data driven setup can be used. In that case a data input can be specified, which is able to handle a collection of data. The number of items in the collection determines the number of Activity instances. This data input can be produced by an input Data Association. The modeler can also configure this loop to control the tokens produced.

- The marker for a Task or Sub-Process that is a multi-instance MUST be a set of three vertical lines.
- If the multi-instance instances are set to be performed in parallel rather than sequential (the isSequential attribute set to false), then the lines of the marker will be vertical (see Figure 10.48).
- If the multi-instance instances are set to be performed in sequence rather than parallel (the isSequential attribute set to true), then the marker will be horizontal (see Figure 10.49).
- The Multi-Instance marker MAY be used in combination with the Compensation marker.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>testBefore: boolean  = false</td>
<td>Flag that controls whether the loop condition is evaluated at the beginning (testBefore = true) or at the end (testBefore = false) of the loop iteration.</td>
</tr>
<tr>
<td>loopMaximum: integer [0..1]</td>
<td>Serves as a cap on the number of iterations.</td>
</tr>
<tr>
<td>loopCondition: Expression [0..1]</td>
<td>A boolean Expression that controls the loop. The Activity will only loop as long as this condition is true. The looping behavior MAY be underspecified, meaning that the modeler can simply document the condition, in which case the loop cannot be formally executed.</td>
</tr>
</tbody>
</table>

Table 10.28 – StandardLoopCharacteristics attributes and model associations

Figure 10.48 - Activity Multi-Instance marker for parallel instances

Figure 10.49 - Activity Multi-Instance marker for sequential instances
The MultiInstanceLoopCharacteristics element inherits the attributes and model associations of BaseElement (see Table 8.5), through its relationship to LoopCharacteristics. Table 10.29 presents the additional attributes and model associations for the MultiInstanceLoopCharacteristics element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>isSequential</strong>: boolean = false</td>
<td>This attribute is a flag that controls whether the Activity instances will execute sequentially or in parallel.</td>
</tr>
<tr>
<td><strong>loopCardinality</strong>: Expression [0..1]</td>
<td>A numeric Expression that controls the number of Activity instances that will be created. This Expression MUST evaluate to an integer. This MAY be underspecified, meaning that the modeler MAY simply document the condition. In such a case the loop cannot be formally executed. In order to initialize a valid multi-instance, either the loopCardinality Expression or the loopDataInput MUST be specified.</td>
</tr>
<tr>
<td><strong>loopDataInputRef</strong>: ItemAwareElement [0..1]</td>
<td>This ItemAwareElement is used to determine the number of Activity instances, one Activity instance per item in the collection of data stored in that ItemAwareElement element. For Tasks it is a reference to a Data Input which is part of the Activity’s InputOutputSpecification. For Sub-Processes it is a reference to a collection-valued Data Object in the context that is visible to the Sub-Processes. In order to initialize a valid multi-instance, either the loopCardinality Expression or the loopDataInput MUST be specified.</td>
</tr>
<tr>
<td><strong>loopDataOutputRef</strong>: ItemAwareElement [0..1]</td>
<td>This ItemAwareElement specifies the collection of data, which will be produced by the multi-instance. For Tasks it is a reference to a Data Output which is part of the Activity’s InputOutputSpecification. For Sub-Processes it is a reference to a collection-valued Data Object in the context that is visible to the Sub-Processes.</td>
</tr>
<tr>
<td><strong>inputDataItem</strong>: DataInput [0..1]</td>
<td>A Data Input, representing for every Activity instance the single item of the collection stored in the loopDataInput. This Data Input can be the source of DataInputAssociation to a data input of the Activity’s InputOutputSpecification. The type of this Data Input MUST the scalar of the type defined for the loopDataInput.</td>
</tr>
<tr>
<td><strong>outputDataItem</strong>: DataOutput [0..1]</td>
<td>A Data Output, representing for every Activity instance the single item of the collection stored in the loopDataOutput. This Data Output can be the target of DataOutputAssociation to a data output of the Activity’s InputOutputSpecification. The type of this Data Output MUST the scalar of the type defined for the loopDataOutput.</td>
</tr>
</tbody>
</table>
Table 10.30 lists all instance attributes available at runtime. For each instance of the Multi-Instance Activity (outer instance), there exists a number of generated (inner) instances of the Activity at runtime.

| behavior: MultiInstanceBehavior = all { None | One | All | Complex } | The attribute behavior acts as a shortcut for specifying when events SHALL be thrown from an Activity instance that is about to complete. It can assume values of None, One, All, and Complex, resulting in the following behavior: |
|---|---|
| None: the EventDefinition which is associated through the noneEvent association will be thrown for each instance completing. |
| One: the EventDefinition referenced through the oneEvent association will be thrown upon the first instance completing. |
| All: no Event is ever thrown; a token is produced after completion of all instances. |
| Complex: the complexBehaviorDefinitions are consulted to determine if and which Events to throw. |

For the behaviors of none and one, a default SignalEventDefinition will be thrown which automatically carries the current runtime attributes of the MI Activity. Any thrown Events can be caught by boundary Events on the Multi-Instance Activity.

| complexBehaviorDefinition: ComplexBehaviorDefinition [0..*] | Controls when and which Events are thrown in case behavior is set to complex. |
| completionCondition: Expression [0..1] | This attribute defines a boolean Expression that when evaluated to true, cancels the remaining Activity instances and produces a token. |
| oneBehaviorEventRef: EventDefinition [0..1] | The EventDefinition which is thrown when behavior is set to one and the first internal Activity instance has completed. |
| noneBehaviorEventRef: EventDefinition [0..1] | The EventDefinition which is thrown when the behavior is set to none and an internal Activity instance has completed. |
Complex Behavior Definition

This element controls when and which Events are thrown in case behavior of the Multi-Instance Activity is set to complex.

The ComplexBehaviorDefinition element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 10.31 presents the additional attributes and model associations for the ComplexBehaviorDefinition element.

### Table 10.30 – Multi-instance Activity instance attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>loopCounter</strong>: integer</td>
<td>This attribute is provided for each generated (inner) instance of the Activity. It contains the sequence number of the generated instance, i.e., if this value of some instance in n, the instance is the n-th instance that was generated.</td>
</tr>
<tr>
<td><strong>numberOfInstances</strong>: integer</td>
<td>This attribute is provided for the outer instance of the Multi-Instance Activity only. This attribute contains the total number of inner instances created for the Multi-Instance Activity.</td>
</tr>
<tr>
<td><strong>numberOfActiveInstances</strong>: integer</td>
<td>This attribute is provided for the outer instance of the Multi-Instance Activity only. This attribute contains the number of currently active inner instances for the Multi-Instance Activity. In case of a sequential Multi-Instance Activity, this value can’t be greater than 1. For parallel Multi-Instance Activities, this value can’t be greater than the value contained in numberOfInstances.</td>
</tr>
<tr>
<td><strong>numberOfCompletedInstances</strong>: integer</td>
<td>This attribute is provided for the outer instance of the Multi-Instance Activity only. This attribute contains the number of already completed inner instances for the Multi-Instance Activity.</td>
</tr>
<tr>
<td><strong>numberOfTerminatedInstances</strong>: integer</td>
<td>This attribute is provided for the outer instance of the Multi-Instance Activity only. This attribute contains the number of terminated inner instances for the Multi-Instance Activity. The sum of numberOfTerminatedInstances, numberOfCompletedInstances, and numberOfActiveInstances always sums up to numberOfInstances.</td>
</tr>
</tbody>
</table>
### 10.2.9 XML Schema for Activities

**Table 10.32 – Activity XML schema**

```xml
<xsd:element name="activity" type="tActivity"/>
<xsd:complexType name="tActivity" abstract="true">
   <xsd:complexContent>
      <xsd:extension base="tFlowNode">
         <xsd:sequence>
            <xsd:element ref="ioSpecification" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="property" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element ref="dataInputAssociation" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element ref="dataOutputAssociation" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element ref="resourceRole" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element ref="loopCharacteristics" minOccurs="0"/>
         </xsd:sequence>
         <xsd:attribute name="isForCompensation" type="xsd:boolean" default="false"/>
         <xsd:attribute name="startQuantity" type="xsd:integer" default="1"/>
         <xsd:attribute name="completetionQuantity" type="xsd:integer" default="1"/>
         <xsd:attribute name="default" type="xsd:IDREF" use="optional"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

---

**Table 10.31 – ComplexBehaviorDefinition attributes and model associations**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>condition</strong>: Formal Expression</td>
<td>This attribute defines a boolean Expression that when evaluated to true, cancels the remaining Activity instances and produces a token.</td>
</tr>
<tr>
<td><strong>event</strong>: ImplicitThrowEvent</td>
<td>If the condition is true, this identifies the Event that will be thrown (to be caught by a boundary Event on the Multi-Instance Activity).</td>
</tr>
</tbody>
</table>
Table 10.33 – AdHocSubProcess XML schema

```xml
<xsd:element name="AdHocSubProcess" type="tAdHocSubProcess" substitutionGroup="flowElement"/>
<xsd:complexType name="tAdHocSubProcess">
    <xsd:complexContent>
        <xsd:extension base="tSubProcess">
            <xsd:sequence>
                <xsd:element name="completionCondition" type="tExpression" minOccurs="0" maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="cancelRemainingInstances" type="xsd:boolean" default="true"/>
            <xsd:attribute name="ordering" type="tAdHocOrdering" default="Parallel"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

```xml
<xsd:simpleType name="tAdHocOrdering">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Parallel"/>
        <xsd:enumeration value="Sequential"/>
    </xsd:restriction>
</xsd:simpleType>
```

Table 10.34 – BusinessRuleTask XML schema

```xml
<xsd:element name="businessRuleTask" type="tBusinessRuleTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tBusinessRuleTask">
    <xsd:complexContent>
        <xsd:extension base="tTask">
            <xsd:attribute name="implementation" type="tImplementation" default="##unspecified"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```
Table 10.35 – CallableElement XML schema

```xml
<xsd:element name="callableElement" type="tCallableElement"/>
<xsd:complexType name="tCallableElement">
   <xsd:complexContent>
      <xsd:extension base="tRootElement">
         <xsd:sequence>
            <xsd:element name="supportedInterfaceRef" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element ref="ioSpecification" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="ioBinding" minOccurs="0" maxOccurs="unbounded"/>
         </xsd:sequence>
         <xsd:attribute name="name" type="xsd:string"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

Table 10.36 – CallActivity XML schema

```xml
<xsd:element name="callActivity" type="tCallActivity" substitutionGroup="flowElement"/>
<xsd:complexType name="tCallActivity">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:attribute name="calledElement" type="xsd:QName" use="optional"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

Table 10.37 – GlobalBusinessRuleTask XML schema

```xml
<xsd:element name="globalBusinessRuleTask" type="tGlobalBusinessRuleTask" substitutionGroup="rootElement"/>
<xsd:complexType name="tGlobalBusinessRuleTask">
   <xsd:complexContent>
      <xsd:extension base="tGlobalTask">
         <xsd:attribute name="implementation" type="tImplementation" default="##unspecified"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```
Table 10.38 – GlobalScriptTask XML schema

```xml
<xsd:element name="globalScriptTask" type="tGlobalScriptTask" substitutionGroup="rootElement"/>
<xsd:complexType name="tGlobalScriptTask">
    <xsd:complexContent>
        <xsd:extension base="tGlobalTask">
            <xsd:sequence>
                <xsd:element ref="script" minOccurs="0" maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="scriptLanguage" type="xsd:anyURI"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.39 – GlobalTask XML schema

```xml
<xsd:element name="globalTask" type="tGlobalTask" substitutionGroup="rootElement"/>
<xsd:complexType name="tGlobalScriptTask">
    <xsd:complexContent>
        <xsd:extension base="tCallableElement">
            <xsd:sequence>
                <xsd:element ref="resourceRole" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.40 – LoopCharacteristics XML schema

```xml
<xsd:element name="loopCharacteristics" type="tLoopCharacteristics"/>
<xsd:complexType name="tLoopCharacteristics" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement"/>
    </xsd:complexContent>
</xsd:complexType>
```
Table 10.41 – MultiInstanceLoopCharacteristics XML schema

```xml
<xsd:element name="multiInstanceLoopCharacteristics" type="tMultiInstanceLoopCharacteristics"
             substitutionGroup="loopCharacteristics"/>
<xsd:complexType name="tMultiInstanceLoopCharacteristics">
    <xsd:complexContent>
        <xsd:extension base="tLoopCharacteristics">
            <xsd:sequence>
                <xsd:element name="loopCardinality" type="tExpression" minOccurs="0"
                             maxOccurs="1"/>
                <xsd:element name="loopDataInputRef" type="xsd:QName" minOccurs="0"
                             maxOccurs="1"/>
                <xsd:element name="loopDataOutputRef" type="xsd:QName" minOccurs="0"
                             maxOccurs="1"/>
                <xsd:element name="inputDataItem" type="tDataInput" minOccurs="0" maxOccurs="1"/>
                <xsd:element name="outputDataItem" type="tDataOutput" minOccurs="0"
                             maxOccurs="1"/>
                <xsd:element ref="complexBehaviorDefinition" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="completionCondition" type="tExpression" minOccurs="0"
                             maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="isSequential" type="xsd:boolean" default="false"/>
            <xsd:attribute name="behavior" type="tMultiInstanceFlowCondition" default="All"/>
            <xsd:attribute name="oneBehaviorEventRef" type="xsd:QName" use="optional"/>
            <xsd:attribute name="noneBehaviorEventRef" type="xsd:QName" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="tMultiInstanceFlowCondition">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="None"/>
        <xsd:enumeration value="One"/>
        <xsd:enumeration value="All"/>
        <xsd:enumeration value="Complex"/>
    </xsd:restriction>
</xsd:simpleType>
```
Table 10.42 – ReceiveTask XML schema

<xsd:element name="receiveTask" type="tReceiveTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tReceiveTask">
  <xsd:complexContent>
    <xsd:extension base="tTask">
      <xsd:attribute name="implementation" type="tImplementation" default="##WebService"/>
      <xsd:attribute name="instantiate" type="xsd:boolean" default="false"/>
      <xsd:attribute name="messageRef" type="xsd:QName" use="optional"/>
      <xsd:attribute name="operationRef" type="xsd:QName" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

Table 10.43 – ResourceRole XML schema

<xsd:element name="resourceRole" type="tResourceRole"/>
<xsd:complexType name="tResourceRole">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:choice>
        <xsd:sequence>
          <xsd:element name="resourceRef" type="xsd:QName" minOccurs="0" maxOccurs="1"/>
          <xsd:element ref="resourceParameterBinding" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:element ref="resourceAssignmentExpression" minOccurs="0" maxOccurs="1"/>
      </xsd:choice>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
Table 10.44 – ScriptTask XML schema

```xml
<xsd:element name="scriptTask" type="tScriptTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tScriptTask">
    <xsd:complexContent>
        <xsd:extension base="tTask">
            <xsd:sequence>
                <xsd:element ref="script" minOccurs="0" maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="scriptFormat" type="xsd:anyURI"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.45 – SendTask XML schema

```xml
<xsd:element name="sendTask" type="tSendTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tSendTask">
    <xsd:complexContent>
        <xsd:extension base="tTask">
            <xsd:attribute name="implementation" type="tImplementation" default="##WebService"/>
            <xsd:attribute name="messageRef" type="xsd:QName" use="optional"/>
            <xsd:attribute name="operationRef" type="xsd:QName" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.46 – ServiceTask XML schema

```xml
<xsd:element name="serviceTask" type="tServiceTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tServiceTask">
    <xsd:complexContent>
        <xsd:extension base="tTask">
            <xsd:attribute name="implementation" type="tImplementation" default="##WebService"/>
            <xsd:attribute name="operationRef" type="xsd:QName" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```
### Table 10.47 – StandardLoopCharacteristics XML schema

```xml
<xsd:element name="standardLoopCharacteristics" type="tStandardLoopCharacteristics"/>
<xsd:complexType name="tStandardLoopCharacteristics">
  <xsd:complexContent>
    <xsd:extension base="tLoopCharacteristics">
      <xsd:sequence>
        <xsd:element name="loopCondition" type="tExpression" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="testBefore" type="xsd:boolean" default="false"/>
      <xsd:attribute name="loopMaximum" type="xsd:integer" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

### Table 10.48 – SubProcess XML schema

```xml
<xsd:element name="subProcess" type="tSubProcess" substitutionGroup="flowElement"/>
<xsd:complexType name="tSubProcess">
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:sequence>
        <xsd:element ref="laneSet" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="flowElement" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="artifact" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="triggeredByEvent" type="xsd:boolean" default="false"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

### Table 10.49 – Task XML schema

```xml
<xsd:element name="task" type="tTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tTask">
  <xsd:complexContent>
    <xsd:extension base="tActivity"/>
  </xsd:complexContent>
</xsd:complexType>
```
A traditional requirement of Process modeling is to be able to model the items (physical or information items) that are created, manipulated, and used during the execution of a Process. An important aspect of this is the ability to capture the structure of that data and to query or manipulate that structure.

BPMN does not itself provide a built-in model for describing structure of data or an Expression language for querying that data. Instead it formalizes hooks that allow for externally defined data structures and Expression languages. In addition, BPMN allows for the co-existence of multiple data structure and Expression languages within the same model. The compatibility and verification of these languages is outside the scope of this specification and becomes the responsibility of the tool vendor.

BPMN designates XML Schema and XPath as its default data structure and Expression languages respectively, but vendors are free to substitute their own languages.

### 10.3.1 Data Modeling

A traditional requirement of Process modeling is to be able to model the items (physical or information items) that are created, manipulated, and used during the execution of a Process.

This requirement is realized in BPMN through various constructs: Data Objects, ItemDefinition, Properties, Data Inputs, Data Outputs, Messages, Input Sets, Output Sets, and Data Associations.

#### Item-Aware Elements

Several elements in BPMN are subject to store or convey items during process execution. These elements are referenced generally as “item-aware elements.” This is similar to the variable construct common to many languages. As with variables, these elements have an ItemDefinition.
The data structure these elements hold is specified using an associated ItemDefinition. An ItemAwareElement MAY be underspecified, meaning that the structure attribute of its ItemDefinition is optional if the modeler does not wish to define the structure of the associated data.

The elements in the specification defined as item-aware elements are: **Data Objects, Data Object References, Data Stores, Properties, DataInputs and DataOutputs**.

![Figure 10.50 - ItemAware class diagram](image)

The ItemAwareElement element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 10.51 presents the additional model associations of the ItemAwareElement element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>itemSubjectRef</strong>: ItemDefinition [0..1]</td>
<td>Specification of the items that are stored or conveyed by the ItemAwareElement.</td>
</tr>
<tr>
<td><strong>dataState</strong>: DataState [0..1]</td>
<td>A reference to the DataState, which defines certain states for the data contained in the Item.</td>
</tr>
</tbody>
</table>
Data Objects

The primary construct for modeling data within the Process flow is the DataObject element. A DataObject has a well-defined lifecycle, with resulting access constraints.

DataObject

The Data Object class is an item-aware element. Data Object elements MUST be contained within Process or Sub-Process elements. Data Object elements are visually displayed on a Process diagram. Data Object References are a way to reuse Data Objects in the same diagram. They can specify different states of the same Data Object at different points in a Process. Data Object Reference cannot specify item definitions, and Data Objects cannot specify states. The names of Data Object References are derived by concatenating the name of the referenced Data Object the state of the Data Object Reference in square brackets as follows: <Data Object Name> [ <Data Object Reference State> ].

Figure 10.51 - DataObject class diagram

The DataObject element inherits the attributes and model associations of FlowElement (see Table 8.44) and ItemAwareElement (Table 10.52). Table 10.54 presents the additional attributes of the DataObject element.
The Data Object element inherits the attributes and model associations of ItemAwareElement (Table 10.52) and FlowElement (see Table 8.44). Table 10.53 presents the additional attributes of the Data Object element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>isCollection: boolean = false</td>
<td>Defines if the Data Object represents a collection of elements. It is needed when no itemDefinition is referenced. If an itemDefinition is referenced, then this attribute MUST have the same value as the isCollection attribute of the referenced itemDefinition. The default value for this attribute is false.</td>
</tr>
</tbody>
</table>

The Data Object Reference element inherits the attributes and model associations of ItemAwareElement (Table 10.52) and FlowElement (see Table 8.44). Table 10.53 presents the additional attributes of the Data Object Reference element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataObjectRef: DataObject</td>
<td>The Data Object referenced by the Data Object Reference.</td>
</tr>
</tbody>
</table>

States

Data Object elements can optionally reference a DataState element, which is the state of the data contained in the Data Object (see an example of DataStates used for Data Objects in Figure 7.8). The definition of these states, e.g., possible values and any specific semantic are out of scope of this specification. Therefore, BPMN adopters can use the State element and the BPMN extensibility capabilities to define their states.

The DataState element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 10.54 presents the additional attributes and model associations of the DataObject element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>Defines the name of the DataState.</td>
</tr>
</tbody>
</table>

Data Objects representing a Collection of Data

A DataObject element that references an ItemDefinition marked as collection has to be visualized differently, compared to single instance data structures. The notation looks as follows:

Single instance (see Figure 10.52)
Figure 10.52 – A DataObject Collection (see Figure 10.53)

Figure 10.53 - A DataObject that is a collection

Visual representations of Data Objects

Data Object can appear multiple times in a Process diagram. Each of these appearances references the same Data Object instance. Multiple occurrences of a Data Object in a diagram are allowed to simplify diagram connections.

Lifecycle and Accessibility

The lifecycle of a Data Object is tied to the lifecycle of its parent Process or Sub-Process. When a Process or Sub-Process is instantiated, all Data Objects contained within it are also instantiated. When a Process or Sub-Process instance is disposed, all Data Object instances contained within it are also disposed. At this point the data within these instances are no longer available.

The accessibility of a Data Object is driven by its lifecycle. The data within a Data Object can only be accessed when there is guaranteed to be a live Data Object instance present. As a result, a Data Object can only be accessed by its immediate parent (Process or Sub-Process), or by its sibling Flow Elements and their children, including Data Object References referencing the Data Object.

For example - Consider the follow structure:

Process A
  Data object 1
  Task A
  Sub-process A
    Data object 2
    Task B
  Sub-process B
    Data object 3
    Sub-process C
      Data object 4
      Task C
    Task D

“Data object 2” can be accessed by: “Sub-Process A” and “Task B.”

“Data object 3” can be accessed by: “Sub-Process B,” “Sub-Process C,” “Task C,” and “Task D.”

“Data object 4” can be accessed by: “Sub-Process C” and “Task C.”

**Data Stores**

A DataStore provides a mechanism for Activities to retrieve or update stored information that will persist beyond the scope of the Process. The same DataStore can be visualized, through a Data Store Reference, in one or more places in the Process.

The Data Store Reference is an ItemAwareElement and can thus be used as the source or target for a Data Association. When data flows into or out of a Data Store Reference, it is effectively flowing into or out of the DataStore that is being referenced.

The notation looks as follows (see Figure 10.54):

![Figure 10.54 - A Data Store](image-url)
The DataStore element inherits the attributes and model associations of FlowElement (see Table 8.44) through its relationship to RootElement, and ItemAwareElement (see Table 10.51). Table 10.55 presents the additional attributes of the DataStore element.

Table 10.55 – Data Store attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>A descriptive name for the element.</td>
</tr>
<tr>
<td>capacity: integer [0..1]</td>
<td>Defines the capacity of the Data Store. This is not needed if the isUnlimited attribute is set to true.</td>
</tr>
<tr>
<td>isUnlimited: boolean = false</td>
<td>If isUnlimited is set to true, then the capacity of a Data Store is set as unlimited and will override any value of the capacity attribute.</td>
</tr>
</tbody>
</table>

The Data Store Reference element inherits the attributes and model associations of FlowElement (see Table 8.44) and ItemAwareElement (see Table 10.51). Table 10.56 presents the additional model associations of the Data Store Reference element.
Properties

Properties, like Data Objects, are item-aware elements. But, unlike Data Objects, they are not visually displayed on a Process diagram. Certain flow elements MAY contain properties, in particular only Processes, Activities, and Events MAY contain Properties.

The Property class is a DataElement element that acts as a container for data associated with flow elements. Property elements MUST be contained within a FlowElement. Property elements are not visually displayed on a Process diagram.

Table 10.56 – Data Store attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataStoreRef: DataStore</td>
<td>Provides the reference to a global DataStore.</td>
</tr>
</tbody>
</table>

Figure 10.56 - Property class diagram

The Property element inherits the attributes and model associations of ItemAwareElement (Table 10.51). Table 10.54 presents the additional attributes of the Property element.
Lifecycle and Accessibility

The lifecycle of a Property is tied to the lifecycle of its parent Flow Element. When a Flow Element is instantiated, all Properties contained by it are also instantiated. When a Flow Element instance is disposed, all Property instances contained by it are also disposed. At this point the data within these instances are no longer available.

The accessibility of a Property is driven by its lifecycle. The data within a Property can only be accessed when there is guaranteed to be a live Property instance present. As a result, a Property can only be accessed by its parent Process, Sub-Process, or Flow Element. In case the parent is a Process or Sub-Process, then a property can be accessed by the immediate children (including children elements) of that Process or Sub-Process. For example, consider the following structure:

```
Process A
  Task A
  Sub-Process A
    Task B
    Sub-Process B
      Sub-Process C
        Task C
        Task D
```


The Properties of “Sub-Process A” are accessible by: “Sub-Process A” and “Task B.”

The Properties of “Task C” are accessible by: “Task C.”

Data Inputs and Outputs

Activities and Processes often need data in order to execute. In addition they can produce data during or as a result of execution. Data requirements are captured as Data Inputs and InputSets. Data that is produced is captured using Data Outputs and OutputSets. These elements are aggregated in a InputOutputSpecification class.

Certain Activities and CallableElements contain a InputOutputSpecification element to describe their data requirements. Execution semantics are defined for the InputOutputSpecification and they apply the same way to all elements that extend it. Not every Activity type defines inputs and outputs, only Tasks, CallableElements (Global Tasks and Processes) MAY define their data requirements. Embedded Sub-Processes MUST NOT define Data Inputs and Data Outputs directly, however they MAY define them indirectly via MultiInstanceLoopCharacteristics.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>Defines the name of the Property.</td>
</tr>
</tbody>
</table>
The `InputOutputSpecification` element inherits the attributes and model associations of `BaseElement` (see Table 8.5). Figure 10.54 presents the additional attributes and model associations of the `InputOutputSpecification` element.
Data Input

A Data Input is a declaration that a particular kind of data will be used as input of the InputOutputSpecification. There may be multiple Data Inputs associated with an InputOutputSpecification.

The Data Input is an item-aware element. Data Inputs are visually displayed on a Process diagram to show the inputs to the top-level Process or to show the inputs of a called Process (i.e., one that is referenced by a Call Activity, where the Call Activity has been expanded to show the called Process within the context of a calling Process).

- Visualized Data Inputs have the same notation as Data Objects, except that they MUST contain a small, unfilled block arrow (see Figure 10.58).
- Data Inputs MAY have incoming Data Associations:
  - If the Data Input is directly contained by the top-level Process, it MUST not be the target of Data Associations within the underlying model. Only Data Inputs that are contained by Activities or Events MAY be the target of Data Associations in the model.
  - If the Process is being called from a Call Activity, the Data Associations that target the Data Inputs of the Call Activity in the underlying model MAY be visualized such that they connect to the corresponding Data Inputs of the called Process, visually crossing the Call Activity boundary. But note that this is visualization only. In the underlying model, the Data Associations target the Data Inputs of the Call Activity and not the Data Inputs of the called Process.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputSets: InputSet [1..*]</td>
<td>A reference to the InputSets defined by the InputOutputSpecification. Every InputOutputSpecification MUST define at least one InputSet.</td>
</tr>
<tr>
<td>outputSets: OutputSet [1..*]</td>
<td>A reference to the OutputSets defined by the InputOutputSpecification. Every Data Interface MUST define at least one OutputSet.</td>
</tr>
<tr>
<td>dataInputs: DataInput [0..*]</td>
<td>An optional reference to the Data Inputs of the InputOutputSpecification. If the InputOutputSpecification defines no Data Input, it means no data is REQUIRED to start the Activity. This is an ordered set.</td>
</tr>
<tr>
<td>dataOutputs: DataOutput [0..*]</td>
<td>An optional reference to the Data Outputs of the InputOutputSpecification. If the InputOutputSpecification defines no Data Output, it means no data is REQUIRED to finish the Activity. This is an ordered set.</td>
</tr>
</tbody>
</table>
Figure 10.58 - A DataInput

The “optional” attribute defines if a DataInput is valid even if the state is “unavailable.” The default value is false. If the value of this attribute is true, then the execution of the Activity will not begin until a value is assigned to the DataInput element, through the corresponding Data Associations.

States

DataInput elements can optionally reference a DataState element, which is the state of the data contained in the DataInput. The definition of these states, e.g., possible values, and any specific semantics are out of scope of this specification. Therefore, BPMN adopters can use the DataState element and the BPMN extensibility capabilities to define their states.

Figure 10.59 - Data Input class diagram

The DataInput element inherits the attributes and model associations of BaseElement (see Table 8.5) and ItemAwareElement (Table 10.52). Table 10.59 presents the additional attributes and model associations of the DataInput element.
Data Output

A **Data Output** is a declaration that a particular kind of data can be produced as output of the InputOutputSpecification. There MAY be multiple **Data Outputs** associated with a **InputOutputSpecification**.

The **Data Output** is an item-aware element. **Data Output** are visually displayed on a top-level Process diagram to show the outputs of the **Process** (i.e., one that is referenced by a Call Activity, where the **Call Activity** has been expanded to show the called **Process** within the context of a calling Process).

- Visualized **Data Outputs** have the same notation as **Data Objects**, except that they MUST contain a small, filled block arrow (see Figure 10.60).
- **Data Outputs** MAY have **outgoing DataAssociations**.
  - If the **Data Output** is directly contained by the top-level **Process**, it MUST not be the source of **Data Associations** within the underlying model. Only **Data Outputs** that are contained by **Activities** or **Events** MAY be the target of **Data Associations** in the model.
  - If the **Process** is being called from a **Call Activity**, the **Data Associations** that target the **Data Outputs** of the **Call Activity** in the underlying model MAY be visualized such that they connect to the corresponding **Data Outputs** of the called **Process**, visually crossing the **Call Activity** boundary. But note that this is visualization only. In the underlying model, the **Data Associations** originate the **Data Outputs** of the **Call Activity** and not the **Data Outputs** of the called **Process**.

Table 10.59 – **DataInput attributes and model associations**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string [0..1]</td>
<td>A descriptive name for the element.</td>
</tr>
<tr>
<td>inputSetRefs: InputSet [1..*]</td>
<td>A <strong>DataInput</strong> is used in one or more <strong>InputSets</strong>. This attribute is derived from the <strong>InputSets</strong>.</td>
</tr>
<tr>
<td>inputSetWithOptional: InputSet</td>
<td>Each <strong>InputSet</strong> that uses this <strong>DataInput</strong> can determine if the <strong>Activity</strong> can start executing with this <strong>DataInput</strong> state in “unavailable.” This attribute lists those <strong>InputSets</strong>.</td>
</tr>
<tr>
<td>[0..*]</td>
<td></td>
</tr>
<tr>
<td>inputSetWithWhileExecuting: Inputset [0..*]</td>
<td>Each <strong>InputSet</strong> that uses this <strong>DataInput</strong> can determine if the <strong>Activity</strong> can evaluate this <strong>DataInput</strong> while executing. This attribute lists those <strong>InputSets</strong>.</td>
</tr>
<tr>
<td>isCollection: boolean = false</td>
<td>Defines if the <strong>DataInput</strong> represents a collection of elements. It is needed when no <strong>itemDefinition</strong> is referenced. If an <strong>itemDefinition</strong> is referenced, then this attribute MUST have the same value as the <strong>isCollection</strong> attribute of the referenced <strong>itemDefinition</strong>. The default value for this attribute is <strong>false</strong>.</td>
</tr>
</tbody>
</table>
DataOutput elements can optionally reference a DataState element, which is the state of the data contained in the DataOutput. The definition of these states, e.g., possible values, and any specific semantics are out of scope of this specification. Therefore, BPMN adopters can use the DataState element and the BPMN extensibility capabilities to define their states.

The DataOutput element inherits the attributes and model associations of BaseElement (see Table 8.5) and ItemAwareElement (Table 10.52). Table 10.60 presents the additional attributes and model associations of the DataInput element.
The following describes the mapping of data inputs and outputs to the specific Activity and Event implementations.

Service Task Mapping
If the Service Task is associated with an Operation, there MUST be a Message Data Input on the Service Task and it MUST have an itemDefinition equivalent to the one defined by the Message referred to by the inMessageRef attribute of the operation. If the operation defines output Messages, there MUST be a single Data Output and it MUST have an itemDefinition equivalent to the one defined by Message referred to by the outMessageRef attribute of the Operation.

Send Task Mapping
If the Send Task is associated with a Message, there MUST be at most inputSet set and at most one Data Input on the Send Task. If the Data Input is present, it MUST have an itemDefinition equivalent to the one defined by the associated Message. If the Data Input is not present, the Message will not be populated with data at execution time.

Receive Task Mapping
If the Receive Task is associated with a Message, there MUST be at most outputSet set and at most one Data Output on the Receive Task. If the Data Output is present, it MUST have an itemDefinition equivalent to the one defined by the associated Message. If the Data Output is not present, the payload within the Message will not flow out of the Receive Task and into the Process.

User Task Mapping
User Tasks have access to the Data Input, Data Output and the data aware elements available in the scope of the User Task.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string [0..1]</td>
<td>A descriptive name for the element.</td>
</tr>
<tr>
<td>outputSetRefs: OutputSet [1..*]</td>
<td>A DataOutput is used in one or more OutputSets. This attribute is derived from the OutputSets.</td>
</tr>
<tr>
<td>outputSetWithOptional: Output-Set [0..*]</td>
<td>Each OutputSet that uses this DataOutput can determine if the Activity can complete executing without producing this DataInput. This attribute lists those OutputSets.</td>
</tr>
<tr>
<td>outputSetWithWhileExecuting: OutputSet [0..*]</td>
<td>Each OutputSet that uses this DataInput can determine if the Activity can produce this DataOutput while executing. This attribute lists those OutputSets.</td>
</tr>
<tr>
<td>isCollection: boolean = false</td>
<td>Defines if the DataOutput represents a collection of elements. It is needed when no itemDefinition is referenced. If an itemDefinition is referenced, then this attribute MUST have the same value as the isCollection attribute of the referenced itemDefinition. The default value for this attribute is false.</td>
</tr>
</tbody>
</table>
Call Activity Mapping

The DataInputs and DataOutputs of the Call Activity are mapped to the corresponding elements in the CallableElement without any explicit DataAssociation.

Script Task Mapping

Script Tasks have access to the Data Input, Data Output and the data aware elements available in the scope of the Script Task.

Events

If any of the EventDefinitions for the Event is associated with an element that has an ItemDefinition (such as a Message, Escalation, Error, or Signal), the following constraints apply:

- If the Event is associated with multiple EventDefinitions, there MUST be one Data Input (in the case of throw Events) or one Data Output (in the case of catch Event) for each EventDefinition. The order of the EventDefinitions and the order of the Data Inputs/Outputs determine which Data Input/Output corresponds with which EventDefinition.
- For each EventDefinition and Data Input/Output pair, if the Data Input/Output is present, it MUST have an ItemDefinition equivalent to the one defined by the Message, Escalation, Error, or Signal on the associated EventDefinition. In the case of a throw Event, if the Data Input is not present, the Message, Escalation, Error, or Signal will not be populated with data. In the case of a catch Event, if the Data Output is not present, the payload within the Message, Escalation, Error, or Signal will not flow out of the Event and into the Process.

InputSet

An InputSet is a collection of DataInput elements that together define a valid set of data inputs for an InputOutputSpecification. An InputOutputSpecification MUST have at least one InputSet element. An InputSet MAY reference zero or more DataInput elements. A single DataInput MAY be associated with multiple InputSet elements, but it MUST always be referenced by at least one InputSet.

An “empty” InputSet, one that references no DataInput elements, signifies that the Activity requires no data to start executing (this implies that either there are no data inputs or they are referenced by another input set).

InputSet elements are contained by InputOutputSpecification elements; the order in which these elements are included defines the order in which they will be evaluated.
The InputSet element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 10.61 presents the additional attributes and model associations of the InputSet element.

Table 10.61 – InputSet attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string [0..1]</td>
<td>A descriptive name for the input set.</td>
</tr>
<tr>
<td>dataInputRefs: DataInput [0..*]</td>
<td>The DataInput elements that collectively make up this data requirement.</td>
</tr>
<tr>
<td>optionalInputRefs: DataInput [0..*]</td>
<td>The DataInput elements that are a part of the InputSet that can be in the state of “unavailable” when the Activity starts executing. This association MUST NOT reference a DataInput that is not listed in the dataInputRefs.</td>
</tr>
<tr>
<td>whileExecutingInputRefs: DataInput [0..*]</td>
<td>The DataInput elements that are a part of the InputSet that can be evaluated while the Activity is executing. This association MUST NOT reference a DataInput that is not listed in the dataInputRefs.</td>
</tr>
<tr>
<td>outputSetRefs: OutputSet [0..*]</td>
<td>Specifies an Input/Output rule that defines which OutputSet is expected to be created by the Activity when this InputSet became valid. This attribute is paired with the inputSetRefs attribute of OutputSets. This combination replaces the IORules attribute for Activities in BPMN 1.2.</td>
</tr>
</tbody>
</table>

OutputSet

An OutputSet is a collection of DataOutputs elements that together can be produced as output from an Activity or Event. An InputOutputSpecification element MUST define at least OutputSet element. An OutputSet MAY reference zero or more DataOutput elements. A single DataOutput MAY be associated with multiple OutputSet elements, but it MUST always be referenced by at least one OutputSet.
An “empty” OutputSet, one that is associated with no DataOutput elements, signifies that the **ACTIVITY** produces no data.

The implementation of the element where the OutputSet is defined determines the OutputSet that will be produced. So it is up to the **Activity** implementation or the **Event**, to define which OutputSet will be produced.

**Figure 10.63 - OutputSet class diagram**

The OutputSet element inherits the attributes and model associations of **BaseElement** (see Table 8.5). Table 10.62 presents the additional attributes and model associations of the **OutputSet** element.
Table 10.62 – OutputSet attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string [0..1]</td>
<td>A descriptive name for the input set.</td>
</tr>
<tr>
<td>dataOutputRefs: DataOutput [0..*]</td>
<td>The DataOutput elements that MAY collectively be outputted.</td>
</tr>
<tr>
<td>optionalOutputRefs: DataOutput [0..*]</td>
<td>The DataOutput elements that are a part of the OutputSet that do not have to be produced when the Activity completes executing. This association MUST NOT reference a DataOutput that is not listed in the dataOutputRefs.</td>
</tr>
<tr>
<td>whileExecutingOutputRefs: DataOutput [0..*]</td>
<td>The DataOutput elements that are a part of the OutputSet that can be produced while the Activity is executing. This association MUST NOT reference a DataOutput that is not listed in the dataOutputRefs.</td>
</tr>
<tr>
<td>inputSetRefs: InputSet [0..*]</td>
<td>Specifies an Input/Output rule that defines which InputSet has to become valid to expect the creation of this OutputSet. This attribute is paired with the outputSetRefs attribute of InputSets. This combination replaces the IORules attribute for Activities in BPMN 1.2.</td>
</tr>
</tbody>
</table>

**Data Associations**

Data Associations are used to move data between Data Objects, Properties, and inputs and outputs of Activities, Processes, and GlobalTasks. Tokens do not flow along a Data Association, and as a result they have no direct effect on the flow of the Process.

The purpose of retrieving data from Data Objects or Process Data Inputs is to fill the Activities inputs and later push the output values from the execution of the Activity back into Data Objects or Process Data Outputs.

**DataAssociation**

The DataAssociation class is a BaseElement contained by an Activity or Event, used to model how data is pushed into or pulled from item-aware elements. DataAssociation elements have one or more sources and a target; the source of the association is copied into the target.

The ItemDefinition from the sourceRef and targetRef MUST have the same ItemDefinition or the DataAssociation MUST have a transformation Expression that transforms the source ItemDefinition into the target ItemDefinition.
Optionally, **Data Associations** can be visually represented in the diagram by using the Association connector style (see Figure 10.65 and Figure 10.66).

The core concepts of a DataAssociation are that they have sources, a target, and an optional transformation.

When a data association is “executed,” data is copied to the target. What is copied depends if there is a transformation defined or not.

If there is no transformation defined or referenced, then only one source MUST be defined, and the contents of this source will be copied into the target.
If there is a transformation defined or referenced, then this transformation Expression will be evaluated and the result of the evaluation is copied into the target. There can be zero (0) to many sources defined in this case, but there is no requirement that these sources are used inside the Expression.

In any case, sources are used to define if the data association can be “executed,” if any of the sources is in the state of “unavailable,” then the data association cannot be executed, and the Activity or Event where the data association is defined MUST wait until this condition is met.

**Data Associations** are always contained within another element that defines when these data associations are going to be executed. Activities define two sets of data associations, while Events define only one.

For Events, there is only one set, but they are used differently for catch or throw Events. For a catch Event, data associations are used to push data from the Message received into Data Objects and properties. For a throw Event, data associations are used to fill the Message that is being thrown.

As DataAssociations are used in different stages of the Process and Activity lifecycle, the possible sources and targets vary according to that stage. This defines the scope of possible elements that can be referenced as source and target. For example: when an Activity starts executing, the scope of valid targets include the Activity data inputs, while at the end of the Activity execution, the scope of valid sources include Activity data outputs.

The DataAssociation element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 10.63 presents the additional model associations of the DataAssociation element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>transformation</strong>: Expression [0..1]</td>
<td>Specifies an optional transformation Expression. The actual scope of accessible data for that Expression is defined by the source and target of the specific Data Association types.</td>
</tr>
<tr>
<td><strong>assignment</strong>: Assignment [0..*]</td>
<td>Specifies one or more data elements Assignments. By using an Assignment, single data structure elements can be assigned from the source structure to the target structure.</td>
</tr>
<tr>
<td><strong>sourceRef</strong>: ItemAwareElement [0..*]</td>
<td>Identifies the source of the Data Association. The source MUST be an ItemAwareElement.</td>
</tr>
<tr>
<td><strong>targetRef</strong>: ItemAwareElement</td>
<td>Identifies the target of the Data Association. The target MUST be an ItemAwareElement.</td>
</tr>
</tbody>
</table>

**Assignment**

The Assignment class is used to specify a simple mapping of data elements using a specified Expression language.

The default Expression language for all Expressions is specified in the Definitions element, using the expressionLanguage attribute. It can also be overridden on each individual Assignment using the same attribute.

The Assignment element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 10.64 presents the additional attributes of the Assignment element.
DataInputAssociation

The DataInputAssociation can be used to associate an ItemAwareElement element with a DataInput contained in an Activity. The source of such a DataAssociation can be every ItemAwareElement accessible in the current scope, e.g., a Data Object, a Property, or an Expression.

The DataInputAssociation element inherits the attributes and model associations of DataAssociation (see Table 10.64), but does not contain any additional attributes or model associations.

Table 10.64 – Assignment attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>from: Expression</td>
<td>The Expression that evaluates the source of the Assignment.</td>
</tr>
<tr>
<td>to: Expression</td>
<td>The Expression that defines the actual Assignment operation and the target data element.</td>
</tr>
</tbody>
</table>

DataOutputAssociation

The DataOutputAssociation can be used to associate a DataOutput contained within an Activity with any ItemAwareElement accessible in the scope the association will be executed in. The target of such a DataAssociation can be every ItemAwareElement accessible in the current scope, e.g., a Data Object, a Property, or an Expression.

The DataOutputAssociation element inherits the attributes and model associations of DataAssociation (see Table 10.64), but does not contain any additional attributes or model associations.

Data Objects associated with a Sequence Flow

Figure 10.67 repeats Figure 10.66, above, and shows how Data Associations are used to represent inputs and outputs of Activities.

Alternatively, Data Objects MAY be directly associated with a Sequence Flow connector (see Figure 10.68) to represent the same input/output relationships. This is a visual short cut that normalizes two Data Associations (e.g., as seen in Figure 10.67, above): one from an item-aware element (e.g., an Activity) contained by the source of the Sequence Flow, connecting to the Data Object, and the other from the Data Object connecting to a item-aware element contained by the target of the Sequence Flow.
10.3.2 Execution Semantics for Data

When an element that defines an InputOutputSpecification is ready to begin execution by means of Sequence Flow or Event being caught, the inputs of the interface are filled with data coming from elements in the context, such as Data Objects or Properties. The way to represent these assignments is the Data Association elements.

Each defined InputSet element will be evaluated in the order they are included in the InputOutputSpecification. For each InputSet, the data inputs it references will be evaluated if it is valid.

All data associations that define as target the data input will be evaluated, and if any of the sources of the data association is “unavailable,” then the InputSet is “unavailable” and the next InputSet is evaluated.

The first InputSet where all data inputs are “available” (by means of data associations) is used to start the execution of the Activity. If no InputSet is “available,” then the execution will wait until this condition is met.

The time and frequency of when and how often this condition is evaluated is out of scope for this specification. Implementations will wait for the sources of data associations to become available and then re-evaluate the InputSets.

In the case of throw and catch Events, given their nature, the execution semantics for data is different.

When a throw Event is activated, all DataInputAssociations of the event are executed, filling the Data Inputs of the Event. Finally, DataInputs are then copied to the elements thrown by the Event (Messages, Signals, etc.). Since there are no InputSets defined for Events, the execution will never wait.

When a catch Event is activated, Data Outputs of the event are filled with the element that triggered the Event. Then all DataOutputAssociations of the Event are executed. There are no OutputSets defined for Events.

To allow invoking a Process from both a Call Activity and via Message Flow, the Start Event and End Event support an additional case.

In the case of a Start Event, the Data Inputs of the enclosing process are available as targets to the DataOutputAssociations of the Event. This way the Process Data Inputs can be filled using the elements that triggered the Start Event.

In the case of an End Event, the Data Outputs of the enclosing process are available as sources to the DataInputAssociations of the Event. This way the resulting elements of the End Event can use the Process Data Outputs as sources.
Once an InputSet becomes “available,” all Data Associations whose target is any of the Data Inputs of the InputSet are executed. These executions fill the Activity Data Inputs and the execution of the Activity can begin. When an Activity finishes execution, all Data Associations whose sources are any of the Data Outputs of the OutputSet are executed. These executions copy the values from the Data Outputs back to the container’s context (Data Object, Properties, etc.).

**Execution Semantics for DataAssociation**

The execution of any Data Associations MUST follow these semantics:

- If the Data Association specifies a “transformation” Expression, this expression is evaluated and the result is copied to the targetRef. This operation replaces completely the previous value of the targetRef element.

- For each “assignment” element specified:
  - Evaluate the Assignment’s “from” expression and obtain the *source value*.
  - Evaluate the Assignment’s “to” expression and obtain the *target element*. The *target element* can be any element in the context or a sub-element of it (e.g., a DataObject or a sub-element of it).
  - Copy the *source value* to the *target element*.

- If no “transformation” Expression nor any “assignment” elements are defined in the Data Association:
  - Copy the Data Association “sourceRef” value into the “targetRef.” Only one sourceRef element is allowed in this case.

**10.3.3 Usage of Data in XPath Expressions**

BPMN extensibility mechanism enables the usage of various languages for Expressions and queries. This section describes how XPath is used in BPMN. It introduces a mechanism to access BPMN Data Objects, BPMN Properties, and various instance attributes from XPath Expressions.

The accessibility by the Expression language is defined based on the entities accessibility by the Activity that contains the Expression. All elements accessible from the enclosing element of an XPath Expression MUST be made available to the XPath processor.

BPMN Data Objects and BPMN Properties are defined using ItemDefinition. The XPath binding assumes that the ItemDefinition is either an XSD complex type or an XSD element. If XSD element is used, it MUST be manifested as a node-set XPath variable with a single member node. If XSD complex type is used, it MUST be manifested as a node-set XPath variable with one member node containing the anonymous document element that contains the actual value of the BPMN Data Object or Property.

**Access to BPMN Data Objects**

Table 10.65 introduces an XPath function used to access BPMN Data Objects. Argument processName names Process and is of type string. Argument dataObjectName names Data Object and is of type string. It MUST be a literal string.
Because XPath 1.0 functions do not support returning faults, an empty node set is returned in the event of an error.

Access to BPMN Data Input and Data Output

Table 10.66 introduces XPath functions used to access BPMN Data Inputs and BPMN Data Outputs. Argument dataInputName names a Data Input and is of type string. Argument dataOutput names a Data Output and is of type string.

<table>
<thead>
<tr>
<th>XPath Extension Function</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element getDataInput('dataInputName')</td>
<td>This extension function returns the value of the submitted Data Input.</td>
</tr>
<tr>
<td>Element getDataOutput('dataOutputName')</td>
<td>This extension function returns the value of the submitted Data Output.</td>
</tr>
</tbody>
</table>

Access to BPMN Properties

Table 10.67 introduces XPath functions used to access BPMN Properties.

Argument processName names Process and is of type string. Argument propertyName names property and is of type string. Argument activityName names Activity and is of type string. Argument eventName names Event and is of type string. These strings MUST be literal strings. The XPath extension functions return value of the submitted property.

Because XPath 1.0 functions do not support returning faults, an empty node set is returned in the event of an error.
Table 10.67 – XPath Extension Functions for Properties

<table>
<thead>
<tr>
<th>XPath Extension Function</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Element getProcessProperty</strong></td>
<td>This extension function returns value of submitted Process property. Argument processName is optional. If omitted, the Process enclosing the Activity that contains the Expression is assumed. In order to access Properties defined in a parent Process the processName MUST be used. Otherwise it MUST be omitted.</td>
</tr>
<tr>
<td>('processName', 'propertyName')</td>
<td></td>
</tr>
<tr>
<td><strong>Element getActivityProperty</strong></td>
<td>This extension function returns value of submitted Activity property.</td>
</tr>
<tr>
<td>('activityName', 'propertyName')</td>
<td></td>
</tr>
<tr>
<td><strong>Element getEventProperty</strong></td>
<td>This extension function returns value of submitted Event property.</td>
</tr>
<tr>
<td>‘eventName’, ‘propertyName’</td>
<td></td>
</tr>
</tbody>
</table>

For BPMN Instance Attributes

Table 10.68 introduces XPath functions used to access BPMN instance Attributes.

Argument processName names Process and is of type string. Argument attributeName names instance attribute and is of type string. Argument activityNameName names Activity and is of type string. These strings MUST be literal strings.

These functions return value of the submitted instance Activity. Because XPath 1.0 functions do not support returning faults, an empty node set is returned in the event of an error.

Table 10.68 – XPath extension functions for instance attributes

<table>
<thead>
<tr>
<th>XPath Extension Function</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Element getProcessInstanceAttribute</strong></td>
<td>This extension function returns value of submitted Process instance attribute. Argument processName is optional. If omitted, the Process enclosing the Activity that contains the Expression is assumed. In order to access instance Attributes of a parent Process the processName MUST be used. Otherwise it MUST be omitted.</td>
</tr>
<tr>
<td>('processName', 'attributeName')</td>
<td></td>
</tr>
<tr>
<td><strong>Element getChoreographyInstanceAttribute</strong></td>
<td>This extension function returns value of submitted Choreography instance attribute.</td>
</tr>
<tr>
<td>('attributeName')</td>
<td></td>
</tr>
<tr>
<td><strong>Element getActivityInstanceAttribute</strong></td>
<td>This extension function returns value of submitted Activity instance attribute. User Task and loop are examples of Activities.</td>
</tr>
<tr>
<td>('activityName', 'attributeName')</td>
<td></td>
</tr>
</tbody>
</table>
10.3.4 XML Schema for Data

Table 10.69 – Assignment XML schema

```xml
<xsd:element name="assignment" type="tAssignment" />
<xsd:complexType name="tAssignment">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="from" type="tExpression" minOccurs="1" maxOccurs="1"/>
                <xsd:element name="to" type="tExpression" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.70 – DataAssociation XML schema

```xml
<xsd:element name="dataAssociation" type="tDataAssociation" />
<xsd:complexType name="tDataAssociation" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="sourceRef" type="xsd:IDREF" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="targetRef" type="xsd:IDREF" minOccurs="1" maxOccurs="1"/>
                <xsd:element name="transformation" type="tFormalExpression" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="assignment" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.71 – DataInput XML schema

```xml
<xsd:element name="dataInput" type="tDataInput" />
<xsd:complexType name="tDataInput">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:attribute name="name" type="xsd:string" use="optional"/>
            <xsd:attribute name="itemSubjectRef" type="xsd:QName"/>
            <xsd:attribute name="isCollection" type="xsd:boolean" default="false"/>
            <xsd:attribute name="dataState" type="xsd:IDREF"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```
Table 10.72 – DataInputAssociation XML schema

```xml
<xsd:element name="dataInputAssociation" type="tDataInputAssociation"/>
<xsd:complexType name="tDataInputAssociation">
  <xsd:complexContent>
    <xsd:extension base="tDataAssociation"/>
  </xsd:complexContent>
</xsd:complexType>
```

Table 10.73 – DataObject XML schema

```xml
<xsd:element name="dataObject" type="tDataObject"/>
<xsd:complexType name="tDataObject">
  <xsd:complexContent>
    <xsd:extension base="tFlowElement">
      <xsd:sequence>
        <xsd:element ref="dataState" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="itemSubjectRef" type="xsd:QName"/>
      <xsd:attribute name="isCollection" type="xsd:boolean"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 10.74 – DataState XML schema

```xml
<xsd:element name="dataState" type="tDataState"/>
<xsd:complexType name="tDataState">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement">
      <xsd:attribute name="name" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
Table 10.75 – DataOutput XML schema

```xml
<xsd:element name="dataOutput" type="tDataOutput" />
<xsd:complexType name="tDataOutput">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:attribute name="name" type="xsd:string" use="optional"/>
            <xsd:attribute name="itemSubjectRef" type="xsd:QName"/>
            <xsd:attribute name="isCollection" type="xsd:boolean" default="false"/>
            <xsd:attribute name="dataState" type="xsd:IDREF"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.76 – DataOutputAssociation XML schema

```xml
<xsd:element name="dataOutputAssociation" type="tDataOutputAssociation" />
<xsd:complexType name="tDataOutputAssociation">
    <xsd:complexContent>
        <xsd:extension base="tDataAssociation"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.77 – InputOutputSpecification XML schema

```xml
<xsd:element name="ioSpecification" type="tInputOutputSpecification" />
<xsd:complexType name="tInputOutputSpecification">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element ref="dataInput" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="dataOutput" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="inputSet" minOccurs="1" maxOccurs="unbounded"/>
                <xsd:element ref="outputSet" minOccurs="1" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```
Table 10.78 – InputSet XML schema

```xml
<xsd:element name="inputSet" type="tInputSet" />
<xsd:complexType name="tInputSet">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="dataInputRefs" type="xsd:IDREF" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="optionalInputRefs" type="xsd:IDREF" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="whileExecutingInputRefs" type="xsd:IDREF" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="outputSetRefs" type="xsd:IDREF" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.79 – OutputSet XML schema

```xml
<xsd:element name="outputSet" type="tOutputSet" />
<xsd:complexType name="tOutputSet">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="dataOutputRefs" type="xsd:IDREF" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="optionalOutputRefs" type="xsd:IDREF" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="whileExecutingOutputRefs" type="xsd:IDREF" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="inputSetRefs" type="xsd:IDREF" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
### 10.4 Events

An Event is something that “happens” during the course of a Process. These Events affect the flow of the Process and usually have a cause or an impact and in general require or allow for a reaction. The term “event” is general enough to cover many things in a Process. The start of an Activity, the end of an Activity, the change of state of a document, a Message that arrives, etc., all could be considered Events.

Events allow for the description of “event-driven” Processes. In these Processes, there are three main types of Events:

1. **Start Events** (see page 238), which indicate where a Process will start.
2. **End Events** (see page 246), which indicate where a path of a Process will end.
3. **Intermediate Events** (see page 249), which indicate where something happens somewhere between the start and end of a Process.

Within these three types, Events come in two flavors:

1. **Events** that catch a trigger. All Start Events and some Intermediate Events are catching Events.
2. **Events** that throw a Result. All End Events and some Intermediate Events are throwing Events that may eventually be caught by another Event. Typically the trigger carries information out of the scope where the throw Event occurred into the scope of the catching Events. The throwing of a trigger may be either implicit as defined by this standard or an extension to it or explicit by a throw Event.
10.4.1 Concepts

Depending on the type of the Event there are different strategies to forward the trigger to catching Events: publication, direct resolution, propagation, cancellations, and compensations.

With publication a trigger MAY be received by any catching Events in any scope of the system where the trigger is published. Events for which publication is used are grouped to Conversations. Published Events MAY participate in several Conversations. Messages are triggers, which are generated outside of the Pool they are published in. They typically describe B2B communication between different Processes in different Pools. When Messages need to reach a specific Process instance, correlation is used to identify the particular instance. Signals are triggers generated in the Pool they are published. They are typically used for broadcast communication within and across Processes, across Pools, and between Process diagrams.

Timer and Conditional triggers are implicitly thrown. When they are activated they wait for a time based or status based condition respectively to trigger the catch Event.
A trigger that is propagated is forwarded from the location where the Event has been thrown to the innermost enclosing scope instance (e.g., Process level) which has an attached Event being able to catch the trigger. Error triggers are critical and suspend execution at the location of throwing. Escalations are non critical and execution continues at the location of throwing. If no catching Event is found for an error or escalation trigger, this trigger is unresolved. Termination, compensation, and cancellation are directed towards a Process or a specific Activity instance. Termination indicates that all Activities in the Process or Activity should be immediately ended. This includes all instances of multi-instances. It is ended without compensation or Event handling.

Compensation of a successfully completed Activity triggers its compensation handler. The compensation handler is either user defined or implicit. The implicit compensation handler of a Sub Process calls all compensation handlers of its enclosed Activities in the reverse order of Sequence Flow dependencies. If compensation is invoked for an Activity that has not yet completed, or has not completed successfully, nothing happens (in particular, no error is raised).

Cancellation will terminate all running Activities and compensate all successfully completed Activities in the Sub-Process it is applied to. If the Sub-Process is a Transaction, the Transaction is rolled back.

Data Modeling and Events

Some Events (like the Message, Escalation, Error, Signal, and Multiple Event) have the capability to carry data. Data Association is used to push data from a Catch Event to a data element. For such Events, the following constraints apply:

- If the Event is associated with multiple EventDefinitions, there MUST be one Data Input (in the case of throw Events) or one Data Output (in the case of catch Events) for each EventDefinition. The order of the EventDefinitions and the order of the Data Inputs/Outputs determine which Data Input/Output corresponds with which EventDefinition.
- For each EventDefinition and Data Input/Output pair, if the Data Input/Output is present, it MUST have an ItemDefinition equivalent to the one defined by the Message, Escalation, Error, or Signal on the associated EventDefinition. In the case of a throw Event, if the Data Input is not present, the Message, Escalation, Error, or Signal will not be populated with data. In the case of a catch Event, if the Data Output is not present, the payload within the Message, Escalation, Error, or Signal will not flow out of the Event and into the Process.

The execution behavior is then as follows:

- For throw Events: When the Event is activated, the data in the Data Input is assigned automatically to the Message, Escalation, Error, or Signal referenced by the corresponding EventDefinition.
- For catch Events: When the trigger of the Event occurs (for example, the Message is received), the data is assigned automatically to the Data Output that corresponds to the EventDefinition that described that trigger.

Common Event attributes

The Event element inherits the attributes and model associations of FlowElement (see Table 8.44). Table 10.81 presents the additional model associations of the Event element.
Common Catch Event attributes

The `CatchEvent` element inherits the attributes and model associations of `Event` element (see Table 10.81). Table 10.82 presents the additional attributes and model associations of the `CatchEvent` element.

### Table 10.81 – Event model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>properties</code>: Property [0..*]</td>
<td>Modeler-defined properties MAY be added to an <code>Event</code>. These properties are contained within the <code>Event</code>.</td>
</tr>
</tbody>
</table>

### Table 10.82 – CatchEvent attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eventDefinitionRefs</code>: EventDefinition [0..*]</td>
<td>References the reusable <code>EventDefinitions</code> that are triggers expected for a <code>catch Event</code>. Reusable <code>EventDefinitions</code> are defined as top-level elements. These <code>EventDefinitions</code> can be shared by different <code>catch</code> and <code>throw Events</code>.</td>
</tr>
<tr>
<td></td>
<td>• If there is no <code>EventDefinition</code> defined, then this is considered a <code>catch None Event</code> and the <code>Event</code> will not have an internal marker (see Figure 10.91).</td>
</tr>
<tr>
<td></td>
<td>• If there is more than one <code>EventDefinition</code> defined, this is considered a <code>Catch Multiple Event</code> and the <code>Event</code> will have the pentagon internal marker (see Figure 10.90).</td>
</tr>
<tr>
<td><code>eventDefinitions</code>: EventDefinition [0..*]</td>
<td>Defines the event <code>EventDefinitions</code> that are triggers expected for a <code>catch Event</code>. These <code>EventDefinitions</code> are only valid inside the current <code>Event</code>.</td>
</tr>
<tr>
<td></td>
<td>• If there is no <code>EventDefinition</code> defined, then this is considered a <code>catch None Event</code> and the <code>Event</code> will not have an internal marker (see Figure 10.91).</td>
</tr>
<tr>
<td></td>
<td>• If there is more than one <code>EventDefinition</code> defined, this is considered a <code>catch Multiple Event</code> and the <code>Event</code> will have the pentagon internal marker (see Figure 10.90).</td>
</tr>
<tr>
<td><code>dataOutputAssociations</code>: Data OutputAssociation [0..*]</td>
<td>The Data Associations of the <code>catch Event</code>. The dataOutputAssociation of a <code>catch Event</code> is used to assign data from the <code>Event</code> to a data element that is in the scope of the <code>Event</code>. For a <code>catch Multiple Event</code>, multiple Data Associations might be REQUIRED, depending on the individual triggers of the <code>Event</code>.</td>
</tr>
<tr>
<td><code>dataOutputs</code>: DataOutput [0..*]</td>
<td>The Data Outputs for the <code>catch Event</code>. This is an ordered set.</td>
</tr>
</tbody>
</table>
Common Throw Event Attributes

The ThrowEvent element inherits the attributes and model associations of Event element (see Table 10.81). Table 10.83 presents the additional attributes and model associations of the ThrowEvent element.

Table 10.83 – ThrowEvent attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
</table>
| eventDefinitionRefs: EventDefinition [0..*] | References the reusable EventDefinitions that are results expected for a throw Event. Reusable EventDefinitions are defined as top-level elements. These EventDefinitions can be shared by different catch and throw Events.  
  - If there is no EventDefinition defined, then this is considered a throw None Event and the Event will not have an internal marker (see Figure 10.91).  
  - If there is more than one EventDefinition defined, this is considered a throw Multiple Event and the Event will have the pentagon internal marker (see Figure 10.90).  
This is an ordered set. |
| eventDefinitions: EventDefinition [0..*] | Defines the event EventDefinitions that are results expected for a throw Event. These EventDefinitions are only valid inside the current Event.  
  - If there is no EventDefinition defined, this is considered a throw None Event and the Event will not have an Internal marker (see Figure 10.91).  
  - If there is more than one EventDefinition defined, this is considered a throw Multiple Event and the Event will have the pentagon internal marker (see Figure 10.90).  
This is an ordered set. |
| dataInputAssociations: DataInput Association [0..*] | The Data Associations of the throw Event.  
The dataInputAssociation of a throw Event is responsible for the assignment of a data element that is in scope of the Event to the Event data.  
For a throw Multiple Event, multiple Data Associations might be REQUIRED, depending on the individual results of the Event. |

outputSet: OutputSet [0..1] The OutputSet for the catch Event.  
parallelMultiple: boolean = false This attribute is only relevant when the catch Event has more than EventDefinition (Multiple).  
If this value is true, then all of the types of triggers that are listed in the catch Event MUST be triggered before the Process is instantiated.
Implicit Throw Event

A sub-type of throw Event is the ImplicitThrowEvent. This is a non-graphical Event that is used for Multi-Instance Activities (see page 191). The ImplicitThrowEvent element inherits the attributes and model associations of ThrowEvent (see Table 10.84), but does not have any additional attributes or model associations.

10.4.2 Start Event

As the name implies, the Start Event indicates where a particular Process will start. In terms of Sequence Flows, the Start Event starts the flow of the Process, and thus, will not have any incoming Sequence Flows—no Sequence Flow can connect to a Start Event.

The Start Event shares the same basic shape of the Intermediate Event and End Event, a circle with an open center so that markers can be placed within the circle to indicate variations of the Event.

A Start Event is a circle that MUST be drawn with a single thin line (see Figure 10.70).

The use of text, color, size, and lines for a Start Event MUST follow the rules defined in “Use of Text, Color, Size, and Lines in a Diagram” on page 41 with the exception that:

- The thickness of the line MUST remain thin so that the Start Event can be distinguished from the Intermediate and End Events.

Figure 10.70 - Start Event

Throughout this document, we discuss how Sequence Flows are used within a Process. To facilitate this discussion, 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 “traverses” the structure of the Process.

Note – A token does not traverse a Message Flow since it is a Message that is passed down a Message Flow (as the name implies).

Semantics of the Start Event include:

- A Start Event is OPTIONAL: a Process level—a top-level Process, a Sub-Process (embedded), or a Global Process (called Process)—MAY (is NOT REQUIRED to) have a Start Event.

Note – A Process MAY have more than one Process level (i.e., it can include Expanded Sub-Processes or Call Activities that call other Processes). The use of Start and End Events is independent for each level of the Diagram.
◆ If a **Process** is complex and/or the starting conditions are not obvious, then it is **RECOMMENDED** that a **Start Event** be used.

◆ If a **Start Event** is not used, then the implicit **Start Event** for the **Process** SHALL NOT have a **trigger**.

◆ If there is an **End Event**, then there MUST be at least one **Start Event**.

◆ All **Flow Objects** that do not have an **incoming Sequence Flow** (i.e., are not a target of a **Sequence Flow**) SHALL be instantiated when the **Process** is instantiated.

  ◆ Exceptions to this are **Activities** that are defined as being **Compensation Activities** (it has the **Compensation** marker). **Compensation Activities** are not considered a part of the **normal flow** and MUST NOT be instantiated when the **Process** is instantiated. See page 302 for more information on **Compensation Activities**.

  ◆ An exception to this is a catching **Link Intermediate Event**, which is not allowed to have **incoming Sequence Flows**. See page 267 for more information on **Link Intermediate Events**.

  ◆ An exception to this is an **Event Sub-Process**, which is not allowed to have **incoming Sequence Flows** and will only be instantiated when its **Start Event** is triggered. See page 176 for more information on **Event Sub-Processes**.

◆ There MAY be multiple **Start Events** for a given **Process** level.

  ◆ Each **Start Event** is an independent **Event**. That is, a **Process instance** SHALL be generated when the **Start Event** is triggered.

If the **Process** is used as a global **Process** (a callable **Process** that can be invoked from **Call Activities** of other **Processes**) and there are multiple **None Start Events**, then when flow is transferred from the parent **Process** to the global **Process**, only one of the global **Process**’s **Start Events** will be triggered. The `targetRef` attribute of a **Sequence Flow incoming** to the **Call Activity** object can be extended to identify the appropriate **Start Event**.

**Note** – The behavior of **Process** can be harder to understand if there are multiple **Start Events**. It is **RECOMMENDED** that this feature be used sparingly and that the modeler be aware that other readers of the Diagram could have difficulty understanding the intent of the Diagram.

When the **trigger** for a **Start Event** occurs, a new **Process** will be instantiated and a **token** will be generated for each **outgoing Sequence Flow** from that **Event**.

**Start Event Triggers**

**Start Events** can be used for these types of **Processes**:

- **Top-level Processes**
- **Sub-Processes** *(embedded)*
- **Global Process** *(called)*
- **Event Sub-Processes**

The next three sections describe the types of **Start Events** that can be used for each of these three types of **Processes**.
Start Events for Top-level Processes

There are many ways that top-level Processes can be started (instantiated). The trigger for a Start Event is designed to show the general mechanisms that will instantiate that particular Process. There are seven (7) types of Start Events for top-level Processes in BPMN (see Table 10.84): None, Message, Timer, Conditional, Signal, Multiple, and Parallel.

A top-level Process that has at least one None Start Event MAY be called by a Call Activity in another Process. The None Start Event is used for invoking the Process from the Call Activity. All other types of Start Events are only applicable when the Process is used as a top-level Process.

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Description</th>
<th>Marker</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>The None Start Event does not have a defined trigger. There is no specific EventDefinition subclass (see page 260) for None Start Events. If the Start Event has no associated EventDefinition, then the Event MUST be displayed without a marker (see the figure on the right).</td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td>A Message arrives from a Participant and triggers the start of the Process. See page 93 for more details on Messages. If there is only one EventDefinition associated with the Start Event and that EventDefinition is of the subclass MessageEventDefinition, then the Event is a Message Start Event and MUST be displayed with an envelope marker (see the figure to the right). The actual Participant from which the Message is received can be identified by connecting the Event to a Participant using a Message Flow within the definitional Collaboration of the Process – see Table 10.1.</td>
<td></td>
</tr>
<tr>
<td>Timer</td>
<td>A specific time-date or a specific cycle (e.g., every Monday at 9am) can be set that will trigger the start of the Process. If there is only one EventDefinition associated with the Start Event and that EventDefinition is of the subclass TimerEventDefinition, then the Event is a Timer Start Event and MUST be displayed with a clock marker (see the figure to the right).</td>
<td></td>
</tr>
</tbody>
</table>
### Conditional
This type of event is triggered when a condition such as “S&P 500 changes by more than 10% since opening”, or “Temperature above 300C” become true. The condition expression for the Event MUST become false and then true before the Event can be triggered again. The Condition Expression of a Conditional Start Event MUST NOT refer to the data context or instance attribute of the Process (as the Process instance has not yet been created). Instead, it MAY refer to static Process attributes and states of entities in the environment. The specification of mechanisms to access such states is out of scope of the standard.

If there is only one EventDefinition associated with the Start Event and that EventDefinition is of the subclass ConditionalEventDefinition, then the Event is a Conditional Start Event and MUST be displayed with a lined paper marker (see the figure to the right).

### Signal
A Signal arrives that has been broadcast from another Process and triggers the start of the Process. Note that the Signal is not a Message, which has a specific target for the Message. Multiple Processes can have Start Events that are triggered from the same broadcasted Signal. If there is only one EventDefinition associated with the Start Event and that EventDefinition is of the subclass SignalEventDefinition, then the Event is a Signal Start Event and MUST be displayed with a triangle marker (see the figure to the right).

### Multiple
This means that there are multiple ways of triggering the Process. Only one of them is REQUIRED. There is no specific EventDefinition subclass for Multiple Start Events. If the Start Event has more than one associated EventDefinition, then the Event MUST be displayed with the Multiple Event marker (a pentagon—see the upper figure to the right).

### Parallel Multiple
This means that there are multiple triggers REQUIRED before the Process can be instantiated. All of the types of triggers that are listed in the Start Event MUST be triggered before the Process is instantiated. There is no specific EventDefinition subclass for Parallel Multiple Start Events. If the Start Event has more than one associated EventDefinition and the parallelMultiple attribute of the Start Event is true, then the Event MUST be displayed with the Parallel Multiple Event marker (an open plus sign—see the figure to the right).

### Start Events for Sub-Processes
There is only one type of Start Event for Sub-Processes in BPMN (see Figure 10.82): None.
Start Events for Event Sub-Processes

A Start Event can also initiate an inline Event Sub-Process (see page 176). In that case, the same Event types as for boundary Events are allowed (see Table 10.86), namely: Message, Timer, Escalation, Error, Compensation, Conditional, Signal, Multiple, and Parallel.

- An Event Sub-Process MUST have a single Start Event.

Table 10.85 – Sub-Process Start Event Types

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Description</th>
<th>Marker</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>The None Start Event is used for all Sub-Processes, either embedded or called (reusable). Other types of triggers are not used for a Sub-Process, since the flow of the Process (a token) from the parent Process is the trigger of the Sub-Process. If the Sub-Process is called (reusable) and has multiple Start Events, some of the other Start Events MAY have triggers, but these Start Events would not be used in the context of a Sub-Process. When the other Start Events are triggered, they would instantiate top-level Processes.</td>
<td></td>
</tr>
<tr>
<td>Event Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td></td>
</tr>
</tbody>
</table>
| Timer      | If there is only one EventDefinition associated with the Start Event and that EventDefinition is of the subclass TimerEventDefinition, then the Event is a Timer Start Event and uses a clock marker (see the figures to the right).  
- For a Timer Event Sub-Process that interrupts its containing Process, the boundary of the Event is solid (see the upper figure to the right).  
- For a Timer Event Sub-Process that does not interrupt its containing Process, the boundary of the Event is dashed (see the lower figure on the right). |
| Escalation | Escalation Event Sub-Processes implement measures to expedite the completion of a business Activity, should it not satisfy a constraint specified on its execution (such as a time-based deadline).  
The Escalation Start Event is only allowed for triggering an in-line Event Sub-Process.  
If there is only one EventDefinition associated with the Start Event and that EventDefinition is of the subclass EscalationEventDefinition, then the Event is an Escalation Start Event and uses an arrowhead marker (see the figures to the right).  
For an Escalation Event Sub-Process that interrupts its containing Process, the boundary of the Event is solid (see the upper figure to the right).  
For an Escalation Event Sub-Process that does not interrupt its containing Process, the boundary of the Event is dashed (see the lower figure on the right). |
| Error      | The Error Start Event is only allowed for triggering an in-line Event Sub-Process.  
If there is only one EventDefinition associated with the Start Event and that EventDefinition is of the subclass ErrorEventDefinition, then the Event is an Error Start Event and uses a lightning marker (see the figures to the right).  
Given the nature of Errors, an Event Sub-Process with an Error trigger will always interrupt its containing Process. |
| Compensation | The Compensation Start Event is only allowed for triggering an in-line Compensation Event Sub-Process (see “Compensation Handler” on page 303). This type of Event is triggered when compensation occurs.  
If there is only one EventDefinition associated with the Start Event and that EventDefinition is of the subclass CompensationEventDefinition, then the Event is a Compensation Start Event and uses a double triangle marker (see the figure to the right). This Event does not interrupt the Process since the Process has to be completed before this Event can be triggered. |
<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional</td>
<td>If there is only one EventDefinition associated with the Start Event and that EventDefinition is of the subclass ConditionalEventDefinition, then the Event is a Conditional Start Event and uses an lined page marker (see the figures to the right). For a Conditional Event Sub-Process that interrupts its containing Process, then the boundary of the Event is solid (see the upper figure to the right). For a Conditional Event Sub-Process that does not interrupt its containing Process, the boundary of the Event is dashed (see the lower figure on the right).</td>
</tr>
<tr>
<td>Signal</td>
<td>If there is only one EventDefinition associated with the Start Event and that EventDefinition is of the subclass SignalEventDefinition, then the Event is a Signal Start Event and uses an triangle marker (see the figures to the right). For a Signal Event Sub-Process that interrupts its containing Process, then the boundary of the Event is solid (see the upper figure to the right). For a Signal Event Sub-Process that does not interrupt its containing Process, the boundary of the Event is dashed (see the lower figure on the right).</td>
</tr>
<tr>
<td>Multiple</td>
<td>A Multiple Event indicates that that there are multiple ways of triggering the Event Sub-Process. Only one of them is REQUIRED to actually start the Event Sub-Process. There is no specific EventDefinition subclass (see page 260) for Multiple Start Events. If the Start Event has more than one associated EventDefinition, then the Event MUST be displayed with the Multiple Event marker (a pentagon—see the figures on the right). For a Multiple Event Sub-Process that interrupts its containing Process, the boundary of the Event is solid (see the upper figure to the right). For a Multiple Event Sub-Process that does not interrupt its containing Process, the boundary of the Event is dashed (see the lower figure on the right).</td>
</tr>
</tbody>
</table>
Attributes for Start Events

For *Start Events*, the following additional attribute exists:

- The *Start Event* element inherits the attributes and model associations of *CatchEvent* (see Table 10.82). Table 10.87 presents the additional attributes of the *Start Event* element:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>isInterrupting</em>: boolean</td>
<td>This attribute only applies to <em>Start Events</em> of <em>Event Sub-Processes</em>; it is ignored for other <em>Start Events</em>. This attribute denotes whether the <em>Sub-Process</em> encompassing the <em>Event Sub-Process</em> should be cancelled or not. If the encompassing <em>Sub-Process</em> is not cancelled, multiple <em>instances</em> of the <em>Event Sub-Process</em> can run concurrently. This attribute cannot be applied to <em>Error Events</em> (where it’s always <em>true</em>), or <em>Compensation Events</em> (where it doesn’t apply).</td>
</tr>
</tbody>
</table>

Sequence Flow Connections

See “Sequence Flow Connections Rules” on page 42 for the entire set of objects and how they MAY be a *source* or *target* of a *Sequence Flow*.

- A *Start Event* MUST NOT be a target for *Sequence Flows*; it MUST NOT have *incoming Sequence Flows*.
  - An exception to this is when a *Start Event* is used in an Expanded *Sub-Process* and is attached to the boundary of that *Sub-Process*. In this case, a *Sequence Flow* from the higher-level *Process* MAY connect to that *Start Event* in lieu of connecting to the actual boundary of the *Sub-Process*.
- A *Start Event* MUST be a source for a *Sequence Flow*.
- Multiple *Sequence Flows* MAY originate from a *Start Event*. For each *Sequence Flow* that has the *Start Event* as a source, a new parallel path SHALL be generated.
  - The *conditionExpression* attribute for all *outgoing Sequence Flows* MUST be set to *None*.
  - When a *Start Event* is not used, then all Flow Objects that do not have an *incoming Sequence Flow* SHALL be the start of a separate parallel path.
  - Each path will have a separate unique *token* that will traverse the *Sequence Flow*. 

Table 10.86 – Event Sub-Process Start Event Types

| Parallel Multiple | A *Parallel Multiple Event* indicates that there are multiple ways of triggering the *Event Sub-Process*. All of them are REQUIRED to actually start the *Event Sub-Process*. There is no specific *EventDefinition* subclass (see page 260) for *Parallel Multiple Start Events*. If the *Start Event* has more than one associated *EventDefinition* and the *parallelMultiple* attribute of the *Start Event* is *true*, then the *Event* MUST be displayed with the *Parallel Multiple Event* marker (an open plus sign—see the figures to the right). For a *Parallel Multiple Event Sub-Process* that interrupts its containing *Process*, the boundary of the *Event* is solid (see the upper figure to the right). For a *Parallel Multiple Event Sub-Process* that does not interrupt its containing *Process*, the boundary of the *Event* is dashed (see the lower figure on the right). |

Table 10.87 – Event Sub-Process Start Event Types

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>isInterrupting</em>: boolean</td>
<td>This attribute only applies to <em>Start Events</em> of <em>Event Sub-Processes</em>; it is ignored for other <em>Start Events</em>. This attribute denotes whether the <em>Sub-Process</em> encompassing the <em>Event Sub-Process</em> should be cancelled or not. If the encompassing <em>Sub-Process</em> is not cancelled, multiple <em>instances</em> of the <em>Event Sub-Process</em> can run concurrently. This attribute cannot be applied to <em>Error Events</em> (where it’s always <em>true</em>), or <em>Compensation Events</em> (where it doesn’t apply).</td>
</tr>
</tbody>
</table>
Message Flow Connections

Note – All Message Flows MUST connect two separate Pools. They MAY connect to the Pool boundary or to Flow Objects within the Pool boundary. They MUST NOT connect two objects within the same Pool.

See “Message Flow Connection Rules” on page 43 for the entire set of objects and how they MAY be a source or targets of a Message Flow.

◆ A Start Event MAY be the target for a Message Flow; it can have zero (0) or more incoming Message Flows. Each Message Flow targeting a Start Event represents an instantiation mechanism (a trigger) for the Process. Only one of the triggers is REQUIRED to start a new Process.
◆ A Start Event MUST NOT be a source for a Message Flow; it MUST NOT have outgoing Message Flows.

10.4.3 End Event

As the name implies, the End Event indicates where a 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—no Sequence Flow can connect from an End Event.

The End Event shares the same basic shape of the Start Event and Intermediate Event, a circle with an open center so that markers can be placed within the circle to indicate variations of the Event.

◆ An End Event is a circle that MUST be drawn with a single thick line (see Figure 10.71).
◆ The use of text, color, size, and lines for an End Event MUST follow the rules defined in “Use of Text, Color, Size, and Lines in a Diagram” on page 41 with the exception that:

◆ The thickness of the line MUST remain thick so that the End Event can be distinguished from the Intermediate and Start Events.

Figure 10.71 - End Event

To continue discussing how flow proceeds throughout the Process, an End Event consumes a token that had been generated from a Start Event within the same level of Process. If parallel Sequence Flows targets the End Event, then the tokens will be consumed as they arrive. All the tokens that were generated within the Process MUST be consumed by an End Event before the Process has been completed. In other circumstances, if the Process is a Sub-Process, it can be stopped prior to normal completion through interrupting Intermediate Events (See Section 10.2.2, “exception flow,” on page 275 for more details). In this situation the tokens will be consumed by an Intermediate Event attached to the boundary of the Sub-Process.

Semantics of the End Event include:

◆ There MAY be multiple End Events within a single level of a Process.
◆ An End Event is OPTIONAL: a given Process level—a Process or an expanded Sub-Process—MAY (is NOT REQUIRED to) have this shape:

◆ If an End Event is not used, then the implicit End Event for the Process SHALL NOT have a Result.
If there is a **Start Event**, then there MUST be at least one **End Event**.

If the **End Event** is not used, then all **Flow Objects** that do not have any *outgoing Sequence Flow* (i.e., are not a source of a **Sequence Flow**) mark the end of a path in the **Process**. However, the **Process** MUST NOT end until all parallel paths have completed.

**Note** – A **Process** MAY have more than one **Process** level (i.e., it can include **Expanded Sub-Processes** or a **Call Activity** that call other **Processes**). The use of **Start** and **End Events** is independent for each level of the Diagram.

For **Processes** without an **End Event**, a *token* entering a path-ending Flow Object will be consumed when the processing performed by the object is completed (i.e., when the path has completed), as if the *token* had then gone on to reach an **End Event**. When all *tokens* for a given instance of the **Process** are consumed, then the **Process** will reach a state of being completed.

**End Event Results**

There are nine types of **End Events** in BPMN: **None**, **Message**, **Escalation**, **Error**, **Cancel**, **Compensation**, **Signal**, **Terminate**, and **Multiple**. These types define the consequence of reaching an **End Event**. This will be referred to as the **End Event Result**.

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Description</th>
<th>Marker</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>The <strong>None End Event</strong> does not have a defined <em>result</em>. There is no specific <strong>EventDefinition</strong> subclass (see page 260) for <strong>None End Events</strong>. If the <strong>End Event</strong> has no associated <strong>EventDefinition</strong>, then the <strong>Event</strong> will be displayed without a marker (see the figure on the right).</td>
<td><img src="image" alt="None" /></td>
</tr>
<tr>
<td>Message</td>
<td>This type of <strong>End</strong> indicates that a <strong>Message</strong> is sent to a <strong>Participant</strong> at the conclusion of the <strong>Process</strong>. See page 93 for more details on <strong>Messages</strong>. The actual <strong>Participant</strong> from which the <strong>Message</strong> is received can be identified by connecting the <strong>Event</strong> to a <strong>Participant</strong> using a <strong>Message Flow</strong> within the definitional <strong>Collaboration</strong> of the <strong>Process</strong> – see Table 10.1.</td>
<td><img src="image" alt="Message" /></td>
</tr>
<tr>
<td>Error</td>
<td>This type of <strong>End</strong> indicates that a named <strong>Error</strong> should be generated. All currently active threads in the particular <strong>Sub-Process</strong> are terminated as a result. The <strong>Error</strong> will be caught by a <strong>Catch Error Intermediate Event</strong> with the same <strong>errorCode</strong> or no <strong>errorCode</strong> which is on the boundary of the nearest enclosing parent <strong>Activity</strong> (hierarchically). The behavior of the <strong>Process</strong> is unspecified if no <strong>Activity</strong> in the hierarchy has such an <strong>Error Intermediate Event</strong>. The system executing the process can define additional <strong>Error handling</strong> in this case, a common one being termination of the <strong>Process instance</strong>.</td>
<td><img src="image" alt="Error" /></td>
</tr>
</tbody>
</table>
### Table 10.88 - End Event Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Escalation</td>
<td>This type of <strong>End</strong> indicates that an <strong>Escalation</strong> should be triggered. Other active threads are not affected by this and continue to be executed. The <strong>Escalation</strong> will be caught by a <strong>Catch Escalation Intermediate Event</strong> with the same <strong>escalationCode</strong> or no <strong>escalationCode</strong> which is on the boundary of the nearest enclosing parent <strong>Activity</strong> (hierarchically). The behavior of the <strong>Process</strong> is unspecified if no <strong>Activity</strong> in the hierarchy has such an <strong>Escalation Intermediate Event</strong>.</td>
</tr>
<tr>
<td>Cancel</td>
<td>This type of <strong>End</strong> is used within a <strong>Transaction Sub-Process</strong>. It will indicate that the Transaction should be cancelled and will trigger a <strong>Cancel Intermediate Event</strong> attached to the <strong>Sub-Process</strong> boundary. In addition, it will indicate that a <strong>TransactionProtocol Cancel Message</strong> should be sent to any <strong>Entities</strong> involved in the <strong>Transaction</strong>.</td>
</tr>
</tbody>
</table>
| Compensation| This type of **End** indicates that compensation is necessary. If an **Activity** is identified, and it was successfully completed, then that **Activity** will be compensated. The **Activity** MUST be visible from the **Compensation End Event**, i.e., one of the following MUST be true:  
  - The **Compensation End Event** is contained in **normal flow** at the same level of **Sub-Process**.  
  - The **Compensation End Event** is contained in a **Compensation Event Sub-Process** that is contained in the **Sub-Process** containing the **Activity**.  
  - If no **Activity** is identified, all successfully completed **Activities** visible from the **Compensation End Event** are compensated, in reverse order of their **Sequence Flows**. Visible means one of the following:  
    - The **Compensation End Event** is contained in **normal flow** and at the same level of **Sub-Process** as the **Activities**.  
    - The **Compensation End Event** is contained in a **Compensation Event Sub-Process** that is contained in the **Sub-Process** containing the **Activities**.  
To be compensated, an **Activity** MUST have a boundary **Compensation Event** or contain a **Compensation Event Sub-Process**. |
| Signal      | This type of **End** indicates that a **Signal** will be broadcasted when the **End** has been reached. Note that the **Signal**, which is broadcast to any **Process** that can receive the **Signal**, can be sent across **Process** levels or **Pools**, but is not a **Message** (that has a specific **source** and **target**). The attributes of a **Signal** can be found on page 273. |
| Terminate   | This type of **End** indicates that all **Activities** in the **Process** should be immediately ended. This includes all **instances** of **multi-instances**. The **Process** is ended without **compensation** or **event handling**. |
Sequence Flow Connections

See “Sequence Flow Connections Rules” on page 42 for the entire set of objects and how they MAY be a source or target of a Sequence Flow.

- An **End Event** MUST be a target for a Sequence Flow.
- An **End Event** MAY have multiple incoming Sequence Flows.

The Flow MAY come from either alternative or parallel paths. For modeling convenience, each path MAY connect to a separate **End Event** object. The **End Event** is used as a Sink for all tokens that arrive at the **Event**. All tokens that are generated at the **Start Event** for that **Process** MUST eventually arrive at an **End Event**. The **Process** will be in a running state until all tokens are consumed.

- An **End Event** MUST NOT be a source for Sequence Flows; that is, there MUST NOT be outgoing Sequence Flows.
- An exception to this is when an **End Event** is used in an Expanded **Sub-Process** and is attached to the boundary of that **Sub-Process**. In this case, a **Sequence Flow** from the higher-level **Process** MAY connect from that **End Event** in lieu of connecting from the actual boundary of the **Sub-Process**.

Message Flow Connections

See “Message Flow Connection Rules” on page 43 for the entire set of objects and how they MAY be a source or target of a Message Flow.

**Note** – All **Message Flows** MUST connect two separate **Pools**. They MAY connect to the **Pool** boundary or to **Flow Objects** within the **Pool** boundary. They MUST NOT connect two objects within the same **Pool**.

- An **End Event** MUST NOT be the target of a Message Flow; it can have no incoming Message Flows.
- An **End Event** MAY be the source of a Message Flow; it can have zero (0) or more outgoing Message Flows. Each **Message Flow** leaving the **End Event** will have a **Message** sent when the **Event** is triggered.
- The Result attribute of the **End Event** MUST be set to Message or Multiple if there are any outgoing Message Flows.
- The Result attribute of the **End Event** MUST be set to Multiple if there is more than one outgoing Message Flows.

**10.4.4 Intermediate Event**

As the name implies, the **Intermediate Event** indicates where something happens (an **Event**) somewhere between the start and end of a **Process**. It will affect the flow of the **Process**, but will not start or (directly) terminate the **Process**. **Intermediate Events** can be used to:

<table>
<thead>
<tr>
<th>Table 10.88 - End Event Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiple</strong></td>
</tr>
</tbody>
</table>
- Show where **Messages** are expected or sent within the **Process**.
- Show delays are expected within the **Process**.
- Disrupt the normal flow through *exception handling*, or
- Show the extra work needed for *compensation*.

The **Intermediate Event** shares the same basic shape of the **Start Event** and **End Event**, a circle with an open center so that markers can be placed within the circle to indicate variations of the **Event**.

- **An Intermediate Event** is a circle that MUST be drawn with a double thin line (see Figure 10.72).
- The use of text, color, size, and lines for an **Intermediate Event** MUST follow the rules defined in “Use of Text, Color, Size, and Lines in a Diagram” on page 41 with the exception that the thickness of the line MUST remain double so that the **Intermediate Event** can be distinguished from the **Start** and **End Events**.

---

**Figure 10.72 - Intermediate Event**

One use of **Intermediate Events** is to represent exception or *compensation handling*. This will be shown by placing the **Intermediate Event** on the boundary of a **Task** or **Sub-Process** (either collapsed or expanded). The **Intermediate Event** can be attached to any location of the **Activity** boundary and the *outgoing Sequence Flows* can flow in any direction. However, in the interest of clarity of the Diagram, we RECOMMEND that the modeler choose a consistent location on the boundary. For example, if the Diagram orientation is horizontal, then the **Intermediate Events** can be attached to the bottom of the **Activity** and the **Sequence Flows** directed down, then to the right. If the Diagram orientation is vertical, then the **Intermediate Events** can be attached to the left or right side of the **Activity** and the **Sequence Flows** directed to the left or right, then down.

**Intermediate Event Triggers**

There are twelve types of **Intermediate Events** in **BPMN**: **None**, **Message**, **Timer**, **Escalation**, **Error**, **Cancel**, **Compensation**, **Conditional**, **Link**, **Signal**, **Multiple**, and **Parallel Multiple**. Each type of **Intermediate Event** will have a different icon placed in the center of the **Intermediate Event** shape to distinguish one from another.

There are two ways that **Intermediate Events** are used in **BPMN**:

1. **An Intermediate Event** that is placed within the *normal flow* of a **Process** can be used for one of two purposes. The **Event** can respond to (“catch”) the **Event trigger** or the **Event** can be used to set off (“throw”) the **Event trigger**.

2. **An Intermediate Event** that is attached to the boundary of an **Activity** can only be used to “catch” the **Event trigger**.

**Intermediate Events in Normal Flow**

When a *token* arrives at an **Intermediate Event** that is placed within the *normal flow* of a **Process**, one of two things will happen.
• If the **Event** is used to “throw” the **Event trigger**, then **trigger** of the **Event** will immediately occur (e.g., the **Message** will be sent) and the **token** will move down the **outgoing Sequence Flow**.

• If the **Event** is used to “catch” the **Event trigger**, then the **token** will remain at the **Event** until the **trigger** occurs (e.g., the **Message** is received). Then the **token** will move down the **outgoing Sequence Flow**.

Ten of the twelve Intermediate Events can be used in normal flow as shown in Table 10.89.

### Table 10.89 - Intermediate Event Types in Normal Flow

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Description</th>
<th>Marker</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>The None Intermediate Event is only valid in normal flow, i.e., it MAY NOT be used on the boundary of an Activity. Although there is no specific trigger for this Event, it is defined as throw Event. It is used for modeling methodologies that use Events to indicate some change of state in the Process. There is no specific EventDefinition subclass (see page 260) for None Intermediate Events. If the (throw) Intermediate Event has no associated EventDefinition, then the Event MUST be displayed without a marker (see the figure on the right).</td>
<td>Throw</td>
</tr>
<tr>
<td>Message</td>
<td>A Message Intermediate Event can be used to either send a Message or receive a Message. When used to “throw” the Message, the Event marker MUST be filled (see the upper figure on the right). When used to “catch” the Message, then the Event marker MUST be unfilled (see the lower figure on the right). This causes the Process to continue if it was waiting for the Message, or changes the flow for exception handling. The actual Participant from which the Message is received can be identified by connecting the Event to a Participant using a Message Flow within the definitional Collaboration of the Process – see Table 10.1. See page 93 for more details on Messages.</td>
<td>Throw, Catch</td>
</tr>
<tr>
<td>Timer</td>
<td>In normal flow the Timer Intermediate Event acts as a delay mechanism based on a specific time-date or a specific cycle (e.g., every Monday at 9am) can be set that will trigger the Event. This Event MUST be displayed with a clock marker (see the figure on the right).</td>
<td>Catch</td>
</tr>
<tr>
<td>Trigger</td>
<td>Description</td>
<td>Marker</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Escalation</td>
<td>In normal flow, the Escalation Intermediate Event raises an Escalation. Since this is a Throw Event, the arrowhead marker will be filled (see the figure to the right).</td>
<td>Throw</td>
</tr>
</tbody>
</table>
| Compensation | In normal flow, this Intermediate Event indicates that compensation is necessary. Thus, it is used to "throw" the Compensation Event, and the Event marker MUST be filled (see figure on the right). If an Activity is identified, and it was successfully completed, then that Activity will be compensated. The Activity MUST be visible from the Compensation Intermediate Event, i.e., one of the following MUST be true:  
  - The Compensation Intermediate Event is contained in normal flow at the same level of Sub-Process.  
  - The Compensation Intermediate Event is contained in a Compensation Event Sub-Process which is contained in the Sub-Process containing the Activity.  
If no Activity is identified, all successfully completed Activities visible from the Compensation Intermediate Event are compensated, in reverse order of their Sequence Flows. Visible means one of the following:  
  - The Compensation Intermediate Event is contained in normal flow and at the same level of Sub-Process as the Activities.  
  - The Compensation Intermediate Event is contained in a Compensation Event Sub-Process which is contained in the Sub-Process containing the Activities.  
To be compensated, an Activity MUST have a boundary Compensation Event, or contain a Compensation Event Sub-Process. | Throw  |
| Conditional  | This type of Event is triggered when a condition becomes true. A condition is a type of Expression. The attributes of an Expression can be found on page 84.                                                             | Catch  |
### Link

The **Link Intermediate Events** are only valid in *normal flow*, i.e., they MAY NOT be used on the boundary of an Activity. A Link is a mechanism for connecting two sections of a Process. Link Events can be used to create looping situations or to avoid long Sequence Flow lines. Link Event uses are limited to a single Process level (i.e., they cannot link a parent Process with a Sub-Process). Paired Intermediate Events can also be used as “Off-Page Connectors” for printing a Process across multiple pages. They can also be used as generic “Go To” objects within the Process level. There can be multiple *source* Link Events, but there can only be one *target* Link Event.

When used to “throw” to the *target* Link, the Event marker will be filled (see the top figure on the right). When used to “catch” from the *source* Link, the Event marker will be unfilled (see the bottom figure on the right).

### Signal

This type of Event is used for sending or receiving Signals. A Signal is for general communication within and across Process levels, across Pools, and between Business Process Diagrams. A BPMN Signal is similar to a signal flare that shot into the sky for anyone who might be interested to notice and then react. Thus, there is a source of the Signal, but no specific intended target. This type of Intermediate Event can send or receive a Signal if the Event is part of a normal flow. The Event can only receive a Signal when attached to the boundary of an Activity. The Signal Event differs from an Error Event in that the Signal defines a more general, non-error condition for interrupting Activities (such as the successful completion of another Activity) as well as having a larger scope than Error Events. When used to “catch” the Signal, the Event marker will be unfilled (see the middle figure on the right). When used to “throw” the Signal, the Event marker will be filled (see the top figure on the right). The attributes of a Signal can be found on page 273.

### Multiple

This means that there are multiple triggers assigned to the Event. If used within normal flow, the Event can “catch” the trigger or “throw” the triggers. When attached to the boundary of an Activity, the Event can only “catch” the trigger. When used to “catch” the trigger, only one of the assigned triggers is REQUIRED and the Event marker will be unfilled (see the middle figure on the right). When used to “throw” the trigger (the same as a Multiple End Event), all the assigned triggers will be thrown and the Event marker will be filled (see the top figure on the right).

There is no specific EventDefinition subclass (see page 260) for Multiple Intermediate Events. If the Intermediate Event has more than one associated EventDefinition, then the Event will be displayed with the Multiple Event marker.
Parallel Multiple | This means that there are multiple *triggers* assigned to the **Event**. If used within *normal flow*, the **Event** can only "catch" the *trigger*. When attached to the boundary of an **Activity**, the **Event** can only "catch" the *trigger*. Unlike the normal **Multiple Intermediate Event**, all of the assigned *triggers* are REQUIRED for the **Event** to be triggered. The **Event** marker will be an unfilled plus sign (see the figure on the right). There is no specific EventDefinition subclass (see page 260) for **Parallel Multiple Intermediate Events**. If the **Intermediate Event** has more than one associated EventDefinition and the parallelMultiple attribute of the **Intermediate Event** is *true*, then the **Event** will be displayed with the **Parallel Multiple Event** marker.

### Intermediate Events Attached to an Activity Boundary

Table 10.90 describes the **Intermediate Events** that can be attached to the boundary of an **Activity**.

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Description</th>
<th>Marker</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Message</strong></td>
<td>A <strong>Message</strong> arrives from a participant and triggers the <strong>Event</strong>. If a <strong>Message Event</strong> is attached to the boundary of an <strong>Activity</strong>, it will change the <em>normal flow</em> into an exception flow upon being triggered. For a <strong>Message Event</strong> that interrupts the <strong>Activity</strong> to which it is attached, the boundary of the <strong>Event</strong> is solid (see upper figure on the right). Note that if using this notation, the attribute cancelActivity of the <strong>Activity</strong> to which the <strong>Event</strong> is attached is implicitly set to <em>true</em>. For a <strong>Message Event</strong> that does not interrupt the <strong>Activity</strong> to which it is attached, the boundary of the <strong>Event</strong> is dashed (see lower figure on the right). Note that if using this notation, the attribute cancelActivity of the <strong>Activity</strong> to which the <strong>Event</strong> is attached is implicitly set to false. The actual <strong>Participant</strong> from which the <strong>Message</strong> is received can be identified by connecting the <strong>Event</strong> to a <strong>Participant</strong> using a <strong>Message Flow</strong> within the definitional <strong>Collaboration</strong> of the <strong>Process</strong> – see Table 10.1.</td>
<td><img src="image1" alt="Interrupting" /> <img src="image2" alt="Non-Interrupting" /></td>
</tr>
<tr>
<td><strong>Timer</strong></td>
<td>A specific time-date or a specific cycle (e.g., every Monday at 9am) can be set that will trigger the <strong>Event</strong>. If a <strong>Timer Event</strong> is attached to the boundary of an <strong>Activity</strong>, it will change the <em>normal flow</em> into an exception flow upon being triggered. For a <strong>Timer Event</strong> that interrupts the <strong>Activity</strong> to which it is attached, the boundary of the <strong>Event</strong> is solid (see upper figure on the right). Note that if using this notation, the attribute cancelActivity of the <strong>Activity</strong> to which the <strong>Event</strong> is attached is implicitly set to <em>true</em>. For a <strong>Timer Event</strong> that does not interrupt the <strong>Activity</strong> to which it is attached, the boundary of the <strong>Event</strong> is dashed (see lower figure on the right). Note that if using this notation, the attribute cancelActivity of the <strong>Activity</strong> to which the <strong>Event</strong> is attached is implicitly set to false.</td>
<td><img src="image3" alt="Interrupting" /> <img src="image4" alt="Non-Interrupting" /></td>
</tr>
<tr>
<td>Event Type</td>
<td>Description</td>
<td>Symbol</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>--------</td>
</tr>
</tbody>
</table>
| Escalation | This type of Event is used for handling a named Escalation. If attached to the boundary of an Activity, the Intermediate Event catches an Escalation. In contrast to an Error, an Escalation by default is assumed to not abort the Activity to which the boundary Event is attached. However, a modeler can decide to override this setting by using the notation described in the following:  
  - For an Escalation Event that interrupts the Activity to which it is attached, the boundary of the Event is solid (see upper figure on the right). Note that if using this notation, the attribute cancelActivity of the Activity to which the Event is attached is implicitly set to true.  
  - For an Escalation Event that does not interrupt the Activity to which it is attached, the boundary of the Event is dashed (see lower figure on the right). Note that if using this notation, the attribute cancelActivity of the Activity to which the Event is attached is implicitly set to false. | ![Interrupting](image1.png) ![Non-Interrupting](image2.png) |
| Error | A catch Intermediate Error Event can only be attached to the boundary of an Activity, i.e., it MAY NOT be used in normal flow. If used in this context, it reacts to (catches) a named Error, or to any Error if a name is not specified. Note that an Error Event always interrupts the Activity to which it is attached, i.e., there is not a non-interrupting version of this Event. The boundary of the Event thus always solid (see figure on the right). | ![Interrupting](image3.png) |
| Cancel | This type of Intermediate Event is used within a Transaction Sub-Process. This type of Event MUST be attached to the boundary of a Sub-Process. It SHALL be triggered if a Cancel End Event is reached within the Transaction Sub-Process. It also SHALL be triggered if a TransactionProtocol "Cancel" Message has been received while the transaction is being performed. Note that a Cancel Event always interrupts the Activity to which it is attached, i.e., there is not a non-interrupting version of this Event. The boundary of the Event thus always solid (see figure on the right). | ![Interrupting](image4.png) |
| Compensation | When attached to the boundary of an Activity, this Event is used to "catch" the Compensation Event, thus the Event marker MUST be unfilled (see figure on the right). The Event will be triggered by a thrown compensation targeting that Activity. When the Event is triggered, the Compensation Activity that is associated to the Event will be performed (see page 302). Note that the interrupting a non-interrupting aspect of other Events does not apply in the case of a Compensation Event. Compensations can only be triggered after completion of the Activity to which they are attached. Thus they cannot interrupt the Activity. The boundary of the Event is always solid. | ![Interrupting](image5.png) |
### Conditional

This type of Event is triggered when a condition becomes true. A condition is a type of Expression. The attributes of an Expression can be found page 84. If a Conditional Event is attached to the boundary of an Activity, it will change the normal flow into an exception flow upon being triggered.

For a Conditional Event that interrupts the Activity to which it is attached, the boundary of the Event is solid (see upper figure on the right). Note that if using this notation, the attribute cancelActivity of the Activity to which the Event is attached is implicitly set to true.

For a Conditional Event that does not interrupt the Activity to which it is attached, the boundary of the Event is dashed (see lower figure on the right). Note that if using this notation, the attribute cancelActivity of the Activity to which the Event is attached is implicitly set to false.

### Signal

The Signal Event can receive a Signal when attached to the boundary of an Activity. In this context, it will change the normal flow into an exception flow upon being triggered. The Signal Event differs from an Error Event in that the Signal defines a more general, non-error condition for interrupting Activities (such as the successful completion of another Activity) as well as having a larger scope than Error Events. When used to “catch” the Signal, the Event marker will be unfilled. The attributes of a Signal can be found on page 273.

For a Signal Event that interrupts the Activity to which it is attached, the boundary of the Event is solid (see upper figure on the right). Note that if using this notation, the attribute cancelActivity of the Activity to which the Event is attached is implicitly set to true.

For a Signal Event that does not interrupt the Activity to which it is attached, the boundary of the Event is dashed (see lower figure on the right). Note that if using this notation, the attribute cancelActivity of the Activity to which the Event is attached is implicitly set to false.
Attributes for Boundary Events

For boundary **Events**, the following additional attributes exist:

- The **BoundaryEvent** element inherits the attributes and model associations of **CatchEvent** (see Table 8.44). Table 8.46 presents the additional attributes and model associations of the **Boundary Event** element.
Table 10.91 – Boundary Event attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>attachedTo: Activity</td>
<td>Denotes the Activity that boundary Event is attached to.</td>
</tr>
<tr>
<td>cancelActivity: boolean</td>
<td>Denotes whether the Activity should be cancelled or not, i.e., whether the boundary catch Event acts as an Error or an Escalation. If the Activity is not cancelled, multiple instances of that handler can run concurrently. This attribute cannot be applied to Error Events (where it’s always true), or Compensation Events (where it doesn’t apply).</td>
</tr>
</tbody>
</table>

Table 10.92 specifies whether the cancel Activity attribute can be set on a boundary Event depending on the EventDefinition it catches.

Table 10.92 – Possible Values of the cancelActivity Attribute

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Possible Values for the cancelActivity Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>N/A as this event cannot be attached to the Activity border.</td>
</tr>
<tr>
<td>Message</td>
<td>True/false</td>
</tr>
<tr>
<td>Timer</td>
<td>True/false</td>
</tr>
<tr>
<td>Escalation</td>
<td>True/false</td>
</tr>
<tr>
<td>Error</td>
<td>True</td>
</tr>
<tr>
<td>Cancel</td>
<td>True</td>
</tr>
<tr>
<td>Compensation</td>
<td>N/A as the scope was already executed and can no longer be canceled when compensation is triggered.</td>
</tr>
<tr>
<td>Conditional</td>
<td>True/false</td>
</tr>
<tr>
<td>Signal</td>
<td>True/false</td>
</tr>
<tr>
<td>Multiple</td>
<td>True/false if all Event triggers allow this option (see this table for details). Otherwise the more restrictive option, i.e., Yes in case any Error or cancel triggers are used.</td>
</tr>
</tbody>
</table>

Activity Boundary Connections

An Intermediate Event can be attached to the boundary of an Activity under the following conditions:

- (One or more) Intermediate Events MAY be attached directly to the boundary of an Activity.
- To be attached to the boundary of an Activity, an Intermediate Event MUST be one of the following triggers (EventDefinition): Message, Timer, Error, Escalation, Cancel, Compensation, Conditional, Signal, Multiple, and Parallel Multiple.
An Intermediate Event with a Cancel trigger MAY be attached to a Sub-Process boundary only if the Transaction attribute of the Sub-Process is set to true.

Sequence Flow Connections
See “Sequence Flow Connections Rules” on page 42 for the entire set of objects and how they MAY be a source or target of a Sequence Flow.

If the Intermediate Event is attached to the boundary of an Activity:

- The Intermediate Event MUST NOT be a target for a Sequence Flow; it cannot have an incoming Sequence Flows.
- Multiple Sequence Flows MAY originate from an Intermediate Event. For each Sequence Flow that has the Intermediate Event as a source, a new parallel path SHALL be generated.
  - An exception to this: an Intermediate Event with a Compensation trigger MUST NOT have an outgoing Sequence Flow (it MAY have an outgoing Association).
- The Intermediate Events with the following triggers (EventDefinition) MAY be used in normal flow: None, Message, Timer, Escalation, Compensation, Conditional, Link, Signal, Multiple, and ParallelMultiple. Thus, the following MUST NOT: Cancel and Error.

If the Intermediate Event is used within normal flow:

- Intermediate Events MUST be a target of a Sequence Flow.

Note – this is a change from BPMN 1.2 semantics, which allowed some Intermediate Events to not have an incoming Sequence Flow.

- An Intermediate Event MAY have multiple incoming Sequence Flows.

Note – If the Event has multiple incoming Sequence Flows, then this is considered uncontrolled flow. This means that when a token arrives from one of the Paths, the Event will be enabled (to catch or throw). It will not wait for the arrival of tokens from the other paths. If another token arrives from the same path or another path, then a separate instance of the Event will be created. If the flow needs to be controlled, then the flow should converge with a Gateway that precedes the Event (see page 287 for more information on Gateways).

- An Intermediate Event MUST be a source for a Sequence Flow.
- Multiple Sequence Flows MAY originate from an Intermediate Event. For each Sequence Flow that has the Intermediate Event as a source, a new parallel path SHALL be generated.
  - An exception to this: a source Link Intermediate Event (as defined below), it is NOT REQUIRED to have an outgoing Sequence Flow.
- A Link Intermediate Event MUST NOT be both a target and a source of a Sequence Flow.

To define the use of a Link Intermediate Event as an “Off-Page Connector” or a “Go To” object:

- A Link Intermediate Event MAY be the target (target Link) or a source (source Link) of a Sequence Flow, but MUST NOT be both a target and a source.
- If there is a source Link, there MUST be a matching target Link (they have the same name).
- There MAY be multiple source Links for a single target Link.
- There MUST NOT be multiple target Links for a single source Link.
Message Flow Connections

See “Message Flow Connection Rules” on page 43 for the entire set of objects and how they MAY be a source or target of a Message Flow.

Note – All Message Flows MUST connect two separate Pools. They MAY connect to the Pool boundary or to Flow Objects within the Pool boundary. They MUST NOT connect two objects within the same Pool.

◆ A Message Intermediate Event MAY be the target for a Message Flow; it can have one incoming Message Flow.
◆ A Message Intermediate Event MAY be a source for a Message Flow; it can have one outgoing Message Flow.
◆ A Message Intermediate Event MAY have an incoming Message Flow or an outgoing Message Flow, but not both.

10.4.5 Event Definitions

Event Definitions refers to the triggers of Catch Events (Start and receive Intermediate Events) and the Results of Throw Events (End Events and send Intermediate Events). The types of Event Definitions are: CancelEventDefinition, CompensationEventDefinition, ConditionalEventDefinition, ErrorEventDefinition, EscalationEventDefinition, MessageEventDefinition, LinkEventDefinition, SignalEventDefinition, TerminateEventDefinition, and TimerEventDefinition (see Table 10.93). A None Event is determined by an Event that does not specify an Event Definition. A Multiple Event is determined by an Event that specifies more than one Event Definition. The different types of Events (Start, End, and Intermediate) utilize a subset of the available types of Event Definitions.
### Table 10.93 – Types of Events and their Markers

<table>
<thead>
<tr>
<th>Types</th>
<th>Start</th>
<th>Intermediate</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top-Level</td>
<td>Event Sub-Process Interrupting</td>
<td>Event Sub-Process Non-Interrupting</td>
</tr>
<tr>
<td>None</td>
<td><img src="none.png" alt="Image" /></td>
<td><img src="none.png" alt="Image" /></td>
<td><img src="none.png" alt="Image" /></td>
</tr>
<tr>
<td>Message</td>
<td><img src="message.png" alt="Image" /></td>
<td><img src="message.png" alt="Image" /></td>
<td><img src="message.png" alt="Image" /></td>
</tr>
<tr>
<td>Timer</td>
<td><img src="timer.png" alt="Image" /></td>
<td><img src="timer.png" alt="Image" /></td>
<td><img src="timer.png" alt="Image" /></td>
</tr>
<tr>
<td>Error</td>
<td><img src="error.png" alt="Image" /></td>
<td><img src="error.png" alt="Image" /></td>
<td><img src="error.png" alt="Image" /></td>
</tr>
<tr>
<td>Escalation</td>
<td><img src="escalation.png" alt="Image" /></td>
<td><img src="escalation.png" alt="Image" /></td>
<td><img src="escalation.png" alt="Image" /></td>
</tr>
<tr>
<td>Cancel</td>
<td><img src="cancel.png" alt="Image" /></td>
<td><img src="cancel.png" alt="Image" /></td>
<td><img src="cancel.png" alt="Image" /></td>
</tr>
<tr>
<td>Compensation</td>
<td><img src="compensation.png" alt="Image" /></td>
<td><img src="compensation.png" alt="Image" /></td>
<td><img src="compensation.png" alt="Image" /></td>
</tr>
<tr>
<td>Conditional</td>
<td><img src="conditional.png" alt="Image" /></td>
<td><img src="conditional.png" alt="Image" /></td>
<td><img src="conditional.png" alt="Image" /></td>
</tr>
<tr>
<td>Link</td>
<td><img src="link.png" alt="Image" /></td>
<td><img src="link.png" alt="Image" /></td>
<td><img src="link.png" alt="Image" /></td>
</tr>
<tr>
<td>Signal</td>
<td><img src="signal.png" alt="Image" /></td>
<td><img src="signal.png" alt="Image" /></td>
<td><img src="signal.png" alt="Image" /></td>
</tr>
<tr>
<td>Terminate</td>
<td><img src="terminate.png" alt="Image" /></td>
<td><img src="terminate.png" alt="Image" /></td>
<td><img src="terminate.png" alt="Image" /></td>
</tr>
<tr>
<td>Multiple</td>
<td><img src="multiple.png" alt="Image" /></td>
<td><img src="multiple.png" alt="Image" /></td>
<td><img src="multiple.png" alt="Image" /></td>
</tr>
</tbody>
</table>
The following sections will present the attributes common to all Event Definitions and the specific attributes for the Event Definitions that have additional attributes. Note that the Cancel and Terminate Event Definitions do not have additional attributes.

**Event Definition Metamodel**

Figure 10.73 shows the class diagram for the abstract class EventDefinition. When one of the EventDefinition sub-types (e.g., TimerEventDefinition) is defined it is contained in Definitions, or a contained EventDefinition contained in a `throw/catch` Event.

The EventDefinition element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to RootElement, but does not contain any additional attributes or model associations.

The ErrorEventDefinition, EscalationEventDefinition, and SignalEventDefinition subclasses comprise of attributes to carry data. The data is defined as part of the Events package. The MessageEventDefinition subclass comprises of an attribute that refers to a Message which is defined as part of the Collaboration package.

The following sections will present the sub-types of EventDefinitions.
Cancel Event

**Cancel Events** are only used in the context of modeling **Transaction Sub-Processes** (see page 178 for more details on **Transactions**). There are two variations: a **catch Intermediate Event** and an **End Event**.

- The **catch Cancel Intermediate Event** MUST only be attached to the boundary of a **Transaction Sub-Process** and, thus, MAY NOT be used in **normal flow**.
- The **Cancel End Event** MUST only be used within a **Transaction Sub-Process** and, thus, MAY NOT be used in any other type of **Sub-Process** or **Process**.

Figure 10.74 shows the variations of **Cancel Events**.

![Cancel Events](image1)

**Figure 10.74 – Cancel Events**

The **CancelEventDefinition** element inherits the attributes and model associations of **BaseElement** (see Table 8.5) through its relationship to the **EventDefinition** element (see page 260).

Compensation Event

**Compensation Events** are used in the context of triggering or handling **compensation** (see page 302 for more details on **compensation**). There are four variations: a **Start Event**, both a **catch** and **throw Intermediate Event**, and an **End Event**.

- The **Compensation Start Event** MAY NOT be used for a **top-level Process**.
- The **Compensation Start Event** MAY be used for an **Event Sub-Process**.
- The **catch Compensation Intermediate Event** MUST only be attached to the boundary of an **Activity** and, thus, MAY NOT be used in **normal flow**.
- The **throw Compensation Intermediate Event** MAY be used in **normal flow**.
- The **Compensation End Event** MAY be used within any **Sub-Process** or **Process**.

Figure 10.75 shows the variations of **Compensation Events**.

![Compensation Events](image2)

**Figure 10.75 – Compensation Events**

Figure 10.76 displays the class diagram for the **CompensationEventDefinition**.
The CompensationEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to the EventDefinition element (see page 260). Table 10.94 presents the additional attributes and model associations of the CompensationEventDefinition element.

Table 10.94 – CompensationEventDefinition attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
</table>
| activityRef: Activity [0..1] | For a Start Event:  
  This Event “catches” the compensation for an Event Sub-Process. No further information is REQUIRED. The Event Sub-Process will provide the Id necessary to match the Compensation Event with the Event that threw the compensation, or the compensation will have been a broadcast.  
  For an End Event:  
  The Activity to be compensated MAY be supplied. If an Activity is not supplied, then the compensation is broadcast to all completed Activities in the current Sub-Process (if present), or the entire Process instance (if at the global level).  
  For an Intermediate Event within normal flow:  
  The Activity to be compensated MAY be supplied. If an Activity is not supplied, then the compensation is broadcast to all completed Activities in the current Sub-Process (if present), or the entire Process instance (if at the global level). This “throws” the compensation.  
  For an Intermediate Event attached to the boundary of an Activity:  
  This Event “catches” the compensation. No further information is REQUIRED. The Activity the Event is attached to will provide the Id necessary to match the Compensation Event with the Event that threw the compensation, or the compensation will have been a broadcast. |
| waitForCompletion: boolean = true | For a throw Compensation Event, this flag determines whether the throw Intermediate Event waits for the triggered compensation to complete (the default), or just triggers the compensation and immediately continues (the BPMN 1.2 behavior). |
Conditional Event

Figure 10.77 shows the variations of Conditional Events.

![Figure 10.77 - Conditional Events](image)

The ConditionalEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to the EventDefinition element (see page 260). Table 10.95 presents the additional model associations of the ConditionalEventDefinition element.

Figure 10.78 displays the class diagram for the ConditionalEventDefinition.

![Figure 10.78 - ConditionalEventDefinition Class Diagram](image)

The ConditionalEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to the EventDefinition element (see page 260). Table 10.95 presents the additional model associations of the ConditionalEventDefinition element.

### Table 10.95 – ConditionalEventDefinition model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition: Expression</td>
<td>The Expression might be underspecified and provided in the form of natural language. For executable Processes (isExecutable = true), if the trigger is Conditional, then a FormalExpression MUST be entered.</td>
</tr>
</tbody>
</table>

Error Event

Figure 10.79 shows the variations of Conditional Events.

![Figure 10.79 - Error Events](image)

Figure 10.80 displays the class diagram for the ErrorEventDefinition.
The ErrorEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to the EventDefinition element (see page 260). Table 10.96 presents the additional attributes and model associations of the ErrorEventDefinition element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>error: Error [0..1]</td>
<td>If the trigger is an Error, then an Error payload MAY be provided.</td>
</tr>
</tbody>
</table>

Escalation Event Definition

Figure 10.81 shows the variations of Escalation Events.

Figure 10.81 – Escalation Events

Figure 10.82 displays the class diagram for the EscalationEventDefinition.
The EscalationEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to the EventDefinition element (see page 260). Table 10.97 presents the additional attributes and model associations of the EscalationEventDefinition element.

### Table 10.97 – EscalationEventDefinition attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>escalationRef: Escalation [0..1]</td>
<td>If the trigger is an Escalation, then an Escalation payload MAY be provided.</td>
</tr>
</tbody>
</table>

#### Link Event Definition

A Link Event is a mechanism for connecting two sections of a Process. Link Events can be used to create looping situations or to avoid long Sequence Flow lines. The use of Link Events is limited to a single Process level (i.e., they cannot link a parent Process with a Sub-Process).

Figure 10.83 shows the variations of Link Events.

**Figure 10.83 – Link Events**

Paired Link Events can also be used as “Off-Page Connectors” for printing a Process across multiple pages. They can also be used as generic “Go To” objects within the Process level. There can be multiple source Link Events, but there can only be one target Link Event. When used to “catch” from the source Link, the Event marker will be unfilled (see Figure 10.84: upper right). When used to “throw” to the target Link, the Event marker will be filled (see Figure 10.84: upper: lower Left).
Since **Process** models often extend beyond the length of one printed page, there is often a concern about showing how **Sequence Flow** connections extend across the page breaks. One solution that is often employed is the use of Off-Page connectors to show where one page leaves off and the other begins. **BPMN** provides **Intermediate Events** of type **Link** for use as Off-Page connectors (see Figure 10.84--Note that the figure shows two different printed pages, not two **Pools** in one diagram). A pair of **Link Events** is used. One of the pair is shown at the end of one page. This **Event** is named and has an **incoming Sequence Flow** and no **outgoing Sequence Flows**. The second **Link Event** is at the beginning of the next page, shares the same name, and has an **outgoing Sequence Flow** and no **incoming Sequence Flow**.

![Diagram](image)

**Figure 10.84 - Link Events Used as Off-Page Connector**

Another way that **Link Events** can be used is as “Go To” objects. Functionally, they would work the same as for Off-Page Connectors (described above), except that they could be used anywhere in the diagram--on the same page or across multiple pages. The general idea is that they provide a mechanism for reducing the length of **Sequence Flow** lines. Some modelers can consider long lines as being hard to follow or trace. Go To Objects can be used to avoid very long
**Sequence Flows** (see Figure 10.85 and Figure 10.86). Both diagrams will behave equivalently. For Figure 10.86, if the “Order Rejected” path is taken from the Decision, then the *token* traversing the **Sequence Flow** would reach the **source Link Event** and then “jump” to the **target Link Event** and continue down the **Sequence Flow**. The **Process** would continue as if the **Sequence Flow** had directly connected the two objects.

![Figure 10.85 - A Process with a long Sequence Flow](image)

Some methodologies prefer that all **Sequence Flows** only move in one direction; that is, forward in time. These methodologies do not allow **Sequence Flows** to connect directly to upstream objects. Some consistency in modeling can be gained by such a methodology, but situations that require looping become a challenge. **Link Events** can be used to make upstream connections and create *loops* without violating the **Sequence Flow** direction restriction (see Figure 10.87).
The `LinkEventDefinition` element inherits the attributes and model associations of `BaseElement` (see Table 8.5) through its relationship to the `EventDefinition` element (see page 260). Table 10.98 presents the additional attributes of the `LinkEventDefinition` element.

### Table 10.98 – LinkEventDefinition attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>If the <code>trigger</code> is a <code>Link</code>, then the <code>name</code> MUST be entered.</td>
</tr>
<tr>
<td>sources: LinkEventDefinition [1..*]</td>
<td>Used to reference the corresponding ‘catch’ or ‘target’ LinkEventDefinition, when this LinkEventDefinition represents a ‘throw’ or ‘source’ LinkEventDefinition.</td>
</tr>
<tr>
<td>target: LinkEventDefinition [1]</td>
<td>Used to reference the corresponding ‘throw’ or ‘source’ LinkEventDefinition, when this LinkEventDefinition represents a ‘catch’ or ‘target’ LinkEventDefinition.</td>
</tr>
</tbody>
</table>

**Message Event Definition**

Figure 10.88 shows the variations of **Message Events**.
Figure 10.89 displays the class diagram for the MessageEventDefinition.

![MessageEventDefinition Class Diagram](image)

The MessageEventDefinition element inherits the attributes and model associations of BaseEntity (see Table 8.5) through its relationship to the EventDefinition element (see page 260). Table 10.99 presents the additional model associations of the MessageEventDefinition element.

### Table 10.99 – MessageEventDefinition model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>messageRef: Message</td>
<td>The Message MUST be supplied (if the isExecutable attribute of the Process is set to true).</td>
</tr>
<tr>
<td>operationRef: Operation</td>
<td>This attribute specifies the Operation that is used by the Message Event. It MUST be specified for executable Processes.</td>
</tr>
</tbody>
</table>

### Multiple Event

For a **Start Event**:

If the trigger is Multiple, there are multiple ways of starting the Process. Only one of them is necessary to trigger the start of the Process. The EventDefinition subclasses will define which triggers apply.

For an **End Event**:

If the Result is Multiple, there are multiple consequences of ending the Process. All of them will occur. The EventDefinition subclasses will define which Results apply.

For an **Intermediate Event** within normal flow:

If the trigger is Multiple, only one EventDefinition is REQUIRED to catch the trigger. When used to throw, all of the EventDefinitions are considered and the subclasses will define which Results apply.
For an **Intermediate Event** attached to the boundary of an **Activity**:

If the _trigger_ is _Multiple_, only one _EventDefinition_ is REQUIRED to “catch” the _trigger_.

Figure 10.90 shows the variations of **Multiple Events**.

![Multiple Events Diagram](image)

**Figure 10.90 – Multiple Events**

**None Event**

**None Events** are **Events** that do not have a defined _EventDefinition_. There are three (3) variations of **None Events**: a **Start Event**, a _catch_ **Intermediate Event**, and an **End Event** (see Figure 10.91).

- The **None Start Event** MAY be used for a _top-level Process_ or any type of _Sub-Process_ (except an _Event Sub-Process_).
- The **None Start Event** MAY NOT be used for an _Event Sub-Process_.
- The _catch_ **None Intermediate Event** MUST only be used in _normal flow_ and, thus, MAY NOT be attached to the boundary of an **Activity**.
- The **None End Event** MAY be used within any _Sub-Process_ or _Process_.

Figure 10.91 shows the variations of **None Events**.

![None Events Diagram](image)

**Figure 10.91 – None Events**

**Parallel Multiple Event**

For a **Start Event**:

- If the _trigger_ is _Multiple_, there are multiple _triggers_ REQUIRED to start the **Process**. All of them are necessary to trigger the start of the **Process**. The _EventDefinition subclasses_ will define which _triggers_ apply. In addition, the _parallelMultiple_ attribute of the **Start Event** MUST be set to _true_.

For an **Intermediate Event** within _normal flow_:

- If the _trigger_ is _Multiple_, all of the defined _EventDefinitions_ are REQUIRED to trigger the **Event**. In addition, the _parallelMultiple_ attribute of the **Intermediate Event** MUST be set to _true_.

For an **Intermediate Event** attached to the boundary of an **Activity**:

- If the _trigger_ is _Multiple_, all of the defined _EventDefinitions_ are REQUIRED to trigger the **Event**. In addition, the _parallelMultiple_ attribute of the **Intermediate Event** MUST be set to _true_.

Figure 10.92 shows the variations of **Parallel Multiple Events**.
Figure 10.92 – Multiple Events

Signal Event

The SignalEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to the EventDefinition element (see page 260). Table 10.100 presents the additional model associations of the ConditionalSignalDefinition element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>signalRef: Signal [0..1]</td>
<td>If the trigger is a Signal, then a Signal is provided.</td>
</tr>
</tbody>
</table>

Terminate Event

Figure 10.95 shows the Terminate Event.
The TerminateEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to the EventDefinition element (see page 260).

Timer Event

Figure 10.96 shows the variations of Timer Events.

The TimerEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8.5) through its relationship to the EventDefinition element (see page 260). Table 10.101 presents the additional model associations of the TimerEventDefinition element.

Table 10.101 – TimerEventDefinition model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeDate:</td>
<td>If the trigger is a Timer, then a timeDate MAY be entered. Timer attributes are mutually exclusive and if any of the other Timer attributes is set, timeDate MUST NOT be set (if the isExecutable attribute of the Process is set to true). The return type of the attribute timeDate MUST conform to the ISO-8601 format for date and time representations.</td>
</tr>
<tr>
<td>timeCycle:</td>
<td>If the trigger is a Timer, then a timeCycle MAY be entered. Timer attributes are mutually exclusive and if any of the other Timer attributes is set, timeCycle MUST NOT be set (if the isExecutable attribute of the Process is set to true). The return type of the attribute timeCycle MUST conform to the ISO-8601 format for recurring time interval representations.</td>
</tr>
<tr>
<td>timeDuration:</td>
<td>If the trigger is a Timer, then a timeDuration MAY be entered. Timer attributes are mutually exclusive and if any of the other Timer attributes is set, timeDuration MUST NOT be set (if the isExecutable attribute of the Process is set to true). The return type of the attribute timeDuration MUST conform to the ISO-8601 format for time interval representations.</td>
</tr>
</tbody>
</table>
10.4.6 Handling Events

BPMN provides advanced constructs for dealing with Events that occur during the execution of a Process (i.e., the “catching” of an Event). Furthermore, BPMN supports the explicit creation of an Event in the Process (i.e., the “throwing” of an Event). Both catching and throwing of an Event as well as the resulting Process behavior is referred to as Event handling. There are three types of Event handlers: those that start a Process, those that are part of the normal Sequence Flow, and those that are attached to Activities, either via boundary Events or via separate inline handlers in case of an Event Sub-Process.

Handling Start Events

There are multiple ways in which a Process can be started. 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. For multiple Start Events, BPMN supports several modeling scenarios that can be applied depending on the scenario.

Exclusive start: the most common scenario for starting a Process is its instantiation by exactly one out of many possible Start Events. Each occurrence of one of these Events will lead to the creation of a new Process instance. The following example shows two Events connected to a single Activity (see Figure 10.97). At runtime, each occurrence of one of the Events will lead to the creation of a new instance of the Process instance and activation of the Activity. Note that a single Multiple Start Event that contains the Message Event Definitions would behave in the same way.

Figure 10.97 - Exclusive start of a Process

<table>
<thead>
<tr>
<th>Message 1</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message 2</td>
<td></td>
</tr>
</tbody>
</table>

A Process can also be started via an Event-Based Gateway, as in the following example (Figure 10.98).
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 an 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.

Event synchronization: if the modeler requires several disjoint Start Events to be merged into a single Process instance, then the following notation MUST be applied (Figure 10.99).

The Parallel Start Event MAY group several disjoint Start Events each of which MUST occur once in order for an instance of the Process to be created. Sequence Flows leaving the Event are then followed as usual.

See page 439 for the execution semantics for the Event Handling of Start Events.

Handling Events within normal Sequence Flow (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.
Handling Events attached to an Activity (Intermediate boundary Events and Event Sub-Processes)

For boundary **Events**, handling consists of consuming the **Event** occurrence and either canceling the **Activity** the **Event** is attached to, followed by normal **Sequence Flows** leaving that **Activity**, or by running an **Event Handler** without canceling the **Activity** (only for **Message**, **Signal**, **Timer** and **Conditional Events**, not for **Error Events**).

An interrupting boundary **Event** is defined by a **true** value of its `cancelActivity` attribute. Whenever the **Event** occurs, the associated **Activity** is terminated. A downstream *token* is then generated, which activates the next element of the **Process** (connected to the **Event** by an unconditional **Sequence Flow** called an *exception flow*).

For non-interrupting boundary **Events**, the `cancelActivity` attribute is set to **false**. Whenever the **Event** occurs, the associated **Activity** continues to be active. As a *token* is generated for the **Sequence Flow** from the boundary **Event** in parallel to the continuing execution of the **Activity**, care MUST be taken when this flow is merged into the main flow of the **Process** – typically it should be ended with its own **End Event**.

The following example shows a fragment (see Figure 10.100) from a trip booking **Process**. It contains a **Sub-Process** that consists of a main part, and three **Event Sub-Processes** to deal with **Events** within the same context: an error **Event Sub-Process** that cancels the **Sub-Process**, a **Message Event Sub-Process** that updates the state of the **Sub-Process** while allowing it to continue, and a **Compensation Event Sub-Process**.
The following example (see Figure 10.101) shows the same fragment of the **Process**, using boundary **Event** handlers rather than inline **Event Sub-Processes**. Note that in this example, the handlers do not have access to the context of the “Booking” **Sub-Process**, as they run outside of it. Therefore, the actually *compensation* logic is shown as a black box.
Note that there is a distinction between interrupting and non-interrupting Events and the handling of these Events, which is described in the sections below. For an interrupting Event (Error, Escalation, Message, Signal, Timer, Conditional, Multiple, and Parallel Multiple), only one Event Sub-Process for the same Event Declaration MUST be modeled. This excludes any further non-interrupting handlers for that Event Declaration.

The reason for this restriction lies in the nature of interrupting Event Sub-Processes and boundary Events. They interrupt normal execution of the parent Activity and after their completion, the parent Activity is immediately terminated. This implies that only one such handler can be executed at a time. However, this does not restrict the modeler in specifying several interrupting handlers, if each handler refers to a different Event Declaration.

For non-interrupting Events (Escalation, Message, Signal, Timer, Conditional, Multiple, and Parallel Multiple), an unlimited number of Event Sub-Processes for the same Event Declaration can be modeled and executed in parallel. At runtime, they will be invoked in a non-deterministic order. The same restrictions apply for boundary Events. During execution of a non-interrupting Event Sub-Process, execution of the parent Activity continues as normal.

If for a given Sub-Process, both an inline Event Sub-Process and a boundary Event handler are modeled that Process the same EventDefinition, the following semantics apply:

◆ If the inline Event Sub-Process “rethrows” the Event after completion, the boundary Event is triggered.
◆ If the inline Event Sub-Process completes without “rethrowing” the Event, the Activity is considered to have completed and normal Sequence Flow resumes. In other terms, the Event Sub-Process “absorbs” the Event.

Interrupting Event Handlers (Error, Escalation, Message, Signal, Timer, Conditional, Multiple, and Parallel Multiple)

Interrupting Event Handlers are those that have the cancelActivity attribute set to true. Whenever the Event occurs, regardless of whether the Event is handled inline or on the boundary, the associated Activity is interrupted. If an inline error handler is specified (in case of a Sub-Process), it is run within the context of that Sub-Process. If a boundary Error Event is present, Sequence Flows from that boundary Event are then followed. The parent Activity is canceled after either the error handler completes or Sequence Flow from the boundary Event is followed.

In the example above, the “Booking” Sub-Process has an Error handler that defines what should happen in case a “Booking” Error occurs within the Sub-Process, namely, the already performed bookings are canceled using compensation. The Error handler is then continued outside the Sub-Process through a boundary Error Event.

Non-interrupting Event Handlers (Escalation, Message, Signal, Timer, Conditional, Multiple, and Parallel Multiple)

Non-interrupting Event Handlers are those that have the cancelActivity attribute set to false.

For Event Sub-Processes, whenever the Event occurs it is consumed and the associated Event Sub-Process is performed. If there are several Events that happen in parallel, then they are handled concurrently, i.e., several Event Sub-Process instances are created concurrently. The non-interrupting Start Event indicates that the Event Sub-Process instance runs concurrently to the Sub-Process proper.

For boundary Events, whenever the Event occurs the handler runs concurrently to the Activity. If an Event Sub-Process is also specified for that Event (in case of a Sub-Process), it is run within the context of that Sub-Process. Then, Sequence Flows from the boundary Event are followed. As a token is generated for the Sequence Flow from the boundary Event in parallel to the continuing execution of the Activity, care MUST be taken when this flow is merged into the main flow of the Process – typically it should be ended with its own End Event.

In the example above, an Event Handler allows to update the credit card information during the “Booking” Sub-Process. It is triggered by a credit card information Message: such a Message can be received whenever the control flow is within the main body of the Sub-Process. Once such a Message is received, the Activities within the corresponding Event Handler run concurrently with the Activities within the body of the Sub-Process.

See “Intermediate Events” on page 440 for the exact semantics of boundary Intermediate Events and “Event Sub-Processes” on page 440 for the operational semantics of non-interrupting Event Sub-Processes.

Handling End Events

For a Terminate End Event, all remaining active Activities within the Process are terminated.

A Cancel End Event is only allowed in the context of a Transaction Sub-Process and, as such, cancels the Sub-Process and aborts an associated Transaction of the Sub-Process.

For all other End Events, the behavior associated with the EventDefinition is performed. When there are no further active Activities, then the Sub-Process or Process instance is completed. See page 443 for exact semantics.
10.4.7 Scopes

A scope describes the context in which execution of an Activity happens. This consists of the set of:

- **Data Objects** available (including DataInput and DataOutput)
- **Events** available for catching or throwing triggers
- **Conversations** going on in that scope

In general, a scope contains exactly one main flow of Activities which is started, when the scope gets activated. Vice versa, all Activities are enclosed by a scope. Scopes are hierarchically nested.

Scopes can have several scope instances at runtime. They are also hierarchically nested according to their generation. In a scope instance several tokens can be active.

Scope instances in turn have a lifecycle containing among others the states:

- Activated
- In execution
- Completed
- In Compensation
- Compensation
- In Error
- In Cancellation
- Cancelled

BPMN has the following model elements with scope characteristics:

- **Choreography**
- **Pool**
- **Sub-Process**
- **Task**
- **Activity**
  - **Multi-instances** body

Scopes are used to define the semantics of

- Visibility of **Data Objects** (including DataInput and DataOutput)
- **Event** resolution
- Starting/stopping of token execution
The **Data Objects, Events, and correlation keys** described by a *scope* can be explicitly modeled or implicitly defined.

### 10.4.8 Events Package XML Schemas

**Table 10.102 – BoundaryEvent XML schema**

```xml
<xsd:element name="boundaryEvent" type="tBoundaryEvent" substitutionGroup="flowElement"/>
<xsd:complexType name="tBoundaryEvent">
    <xsd:complexContent>
        <xsd:extension base="tCatchEvent">
            <xsd:attribute name="cancelActivity" type="xsd:boolean" default="true"/>
            <xsd:attribute name="attachedToRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

**Table 10.103 – CancelEventDefinition XML schema**

```xml
<xsd:element name="cancelEventDefinition" type="tCancelEventDefinition" substitutionGroup="eventDefinition"/>
<xsd:complexType name="tCancelEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition"/>
    </xsd:complexContent>
</xsd:complexType>
```

**Table 10.104 – CatchEvent XML schema**

```xml
<xsd:element name="catchEvent" type="tCatchEvent"/>
<xsd:complexType name="tCatchEvent" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tEvent">
            <xsd:sequence>
                <xsd:element ref="dataOutput" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="dataOutputAssociation" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="outputSet" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="eventDefinition" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="eventDefinitionRef" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="parallelMultiple" type="xsd:boolean" default="false"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```
Table 10.105  – CancelEventDefinition XML schema

```xml
<xsd:element name="cancelEventDefinition" type="tCancelEventDefinition" substitutionGroup="eventDefinition"/>
<xsd:complexType name="tCancelEventDefinition">
  <xsd:complexContent>
    <xsd:extension base="tEventDefinition"/>
  </xsd:complexContent>
</xsd:complexType>
```

Table 10.106  – CompensateEventDefinition XML schema

```xml
<xsd:element name="compensateEventDefinition" type="tCompensateEventDefinition" substitutionGroup="eventDefinition"/>
<xsd:complexType name="tCompensateEventDefinition">
  <xsd:complexContent>
    <xsd:extension base="tEventDefinition">
      <xsd:attribute name="waitForCompletion" type="xsd:boolean"/>
      <xsd:attribute name="activityRef" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 10.107  – ConditionalEventDefinition XML schema

```xml
<xsd:element name="conditionalEventDefinition" type="tConditionalEventDefinition" substitutionGroup="eventDefinition"/>
<xsd:complexType name="tConditionalEventDefinition">
  <xsd:complexContent>
    <xsd:extension base="tEventDefinition">
      <xsd:sequence>
        <xsd:element name="condition" type="tExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 10.108  – ErrorEventDefinition XML schema

```xml
<xsd:element name="errorEventDefinition" type="tErrorEventDefinition" substitutionGroup="eventDefinition"/>
<xsd:complexType name="tErrorEventDefinition">
  <xsd:complexContent>
    <xsd:extension base="tEventDefinition">
      <xsd:attribute name="errorRef" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 10.109  – EscalationEventDefinition XML schema

```xml
<xsd:element name="escalationEventDefinition" type="tEscalationEventDefinition" substitutionGroup="eventDefinition"/>
```
Table 10.110 – Event XML schema

<xsd:element name="event" type="tEvent" substitutionGroup="flowElement"/>
<xsd:complexType name="tEvent" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tFlowNode"/>
    </xsd:complexContent>
</xsd:complexType>

Table 10.111 – EventDefinition XML schema

<xsd:element name="eventDefinition" type="tEventDefinition"/>
<xsd:complexType name="tEventDefinition" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement"/>
    </xsd:complexContent>
</xsd:complexType>

Table 10.112 – ImplicitThrowEvent XML schema

<xsd:element name="implicitThrowEvent" type="tImplicitThrowEvent" substitutionGroup="flowElement"/>
<xsd:complexType name="tImplicitThrowEvent">
    <xsd:complexContent>
        <xsd:extension base="tThrowEvent"/>
    </xsd:complexContent>
</xsd:complexType>

Table 10.113 – IntermediateCatchEvent XML schema

<xsd:element name="intermediateCatchEvent" type="tIntermediateCatchEvent" substitutionGroup="flowElement"/>
<xsd:complexType name="tIntermediateCatchEvent">
    <xsd:complexContent>
        <xsd:extension base="tCatchEvent"/>
    </xsd:complexContent>
</xsd:complexType>

Table 10.114 – IntermediateThrowEvent XML schema

<xsd:element name="intermediateThrowEvent" type="tIntermediateThrowEvent" substitutionGroup="flowElement"/>
Table 10.115 – LinkEventDefinition XML schema

```xml
<xsd:complexType name="tLinkEventDefinition">
  <xsd:complexContent>
    <xsd:extension base="tEventDefinition">
      <xsd:sequence>
        <xsd:element name="source" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="target" type="xsd:QName" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 10.116 – MessageEventDefinition XML schema

```xml
<xsd:complexType name="tMessageEventDefinition">
  <xsd:complexContent>
    <xsd:extension base="tEventDefinition">
      <xsd:sequence>
        <xsd:element name="operationRef" type="xsd:QName" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="messageRef" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 10.117 – Signal XML schema

```xml
<xsd:complexType name="tSignal">
  <xsd:complexContent>
    <xsd:extension base="tRootElement">
      <xsd:attribute name="name" type="xsd:string"/>
      <xsd:attribute name="structureRef" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
Table 10.118 – SignalEventDefinition XML schema

```xml
<xsd:element name="signalEventDefinition" type="tSignalEventDefinition" substitutionGroup="eventDefinition"/>
<xsd:complexType name="tSignalEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition">
            <xsd:attribute name="signalRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.119 – StartEvent XML schema

```xml
<xsd:element name="startEvent" type="tStartEvent" substitutionGroup="flowElement"/>
<xsd:complexType name="tStartEvent">
    <xsd:complexContent>
        <xsd:extension base="tCatchEvent">
            <xsd:attribute name="isInterrupting" type="xsd:boolean" default="true"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.120 – TerminateEventDefinition XML schema

```xml
<xsd:element name="terminateEventDefinition" type="tTerminateEventDefinition" substitutionGroup="eventDefinition"/>
<xsd:complexType name="tTerminateEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.121 – ThrowEvent XML schema

```xml
<xsd:element name="throwEvent" type="tThrowEvent" abstract="true"/>
<xsd:complexType name="tThrowEvent">
    <xsd:complexContent>
        <xsd:extension base="tEvent">
            <xsd:sequence>
                <xsd:element ref="dataInput" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="dataInputAssociation" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="inputSet" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="eventDefinition" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="eventDefinitionRef" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```
Table 10.122 – TimerEventDefinition XML schema

```xml
<xsd:element name="timerEventDefinition" type="tTimerEventDefinition" substitutionGroup="eventDefinition"/>
<xsd:complexType name="tTimerEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition">
            <xsd:choice>
                <xsd:element name="timeDate" type="tExpression" minOccurs="0" maxOccurs="1"/>
                <xsd:element name="timeDuration" type="tExpression" minOccurs="0" maxOccurs="1"/>
                <xsd:element name="timeCycle" type="tExpression" minOccurs="0" maxOccurs="1"/>
            </xsd:choice>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

10.5 Gateways

Gateways are used to control how Sequence Flows interact as they converge and diverge within a Process. If the flow does not need to be controlled, then a Gateway is not needed. The term “Gateway” implies that there is a gating mechanism that either allows or disallows passage through the Gateway—that is, as tokens arrive at a Gateway, they can be merged together on input and/or split apart on output as the Gateway mechanisms are invoked.

A Gateway is a diamond, which has been used in many flow chart notations for exclusive branching and is familiar to most modelers.

- A Gateway is a diamond that MUST be drawn with a single thin line (see Figure 10.102).
- The use of text, color, size, and lines for a Gateway MUST follow the rules defined in “Use of Text, Color, Size, and Lines in a Diagram” on page 41 with the exception that:

![Figure 10.102 – A Gateway](image)

Gateways, like Activities, are capable of consuming or generating additional tokens, effectively controlling the execution semantics of a given Process. The main difference is that Gateways do not represent ‘work’ being done and they are considered to have zero effect on the operational measures of the Process being executed (cost, time, etc.).

Gateways can define all the types of Business Process Sequence Flow behavior: Decisions/branching (exclusive, inclusive, and complex), merging, forking, and joining. Thus, while the diamond has been used traditionally for exclusive decisions, BPMN extends the behavior of the diamonds to reflect any type of Sequence Flow control. Each type of Gateway will have an internal indicator or marker to show the type of Gateway that is being used (see Figure 10.103).
The **Gateway** controls the flow of both diverging and converging **Sequence Flows**. That is, a single **Gateway** could have multiple input and multiple output flows. Modelers and modeling tools might want to enforce a best practice of a **Gateway** only performing one of these functions. Thus, it would take two sequential **Gateways** to first converge and then to diverge the **Sequence Flows**.
Figure 10.104 - Gateway class diagram

**Gateways** are described in this section on an abstract level. The execution semantics of **Gateways** is detailed on page 434.

### 10.5.1 Sequence Flow Considerations

**Note** – Although the shape of a **Gateway** is a diamond, it is not a requirement that *incoming* and *outgoing* **Sequence Flows** MUST connect to the corners of the diamond. **Sequence Flows** can connect to any position on the boundary of the **Gateway** shape.

This section applies to all **Gateways**. Additional **Sequence Flow** Connection rules are specified for each type of **Gateway** in the sections below.

- A **Gateway** MAY be a target for a **Sequence Flow**. It can have zero (0), one (1), or more *incoming* **Sequence Flows**.
- If the **Gateway** does not have an *incoming Sequence Flow*, and there is no **Start Event** for the **Process**, then the **Gateway**'s divergence behavior, depending on the type of **Gateway** (see below), SHALL be performed when the **Process** is instantiated.
A **Gateway** MAY be a source of a **Sequence Flow**; it can have zero, one, or more **outgoing Sequence Flows**.

A **Gateway** MUST have either multiple **incoming Sequence Flows** or multiple **outgoing Sequence Flows** (i.e., it MUST merge or split the flow).

- A **Gateway** with a gatewayDirection of unspecified MAY have both multiple **incoming** and **outgoing Sequence Flows**.
- A **Gateway** with a gatewayDirection of mixed MUST have both multiple **incoming** and **outgoing Sequence Flows**.
- A **Gateway** with a gatewayDirection of converging MUST have multiple **incoming Sequence Flows**, but MUST NOT have multiple **outgoing Sequence Flows**.
- A **Gateway** with a gatewayDirection of diverging MUST have multiple **outgoing Sequence Flows**, but MUST NOT have multiple **incoming Sequence Flows**.

### 10.5.2 Exclusive Gateway

A diverging **Exclusive Gateway (Decision)** is used to create alternative paths within a **Process** flow. This is basically the “diversion point in the road” for a **Process**. For a given instance of the **Process**, only one of the paths can be taken.

A Decision can be thought of as a question that is asked at a particular point in the **Process**. The question has a defined set of alternative answers. Each answer is associated with a condition **Expression** that is associated with a **Gateway**’s **outgoing Sequence Flows**.

- The **Exclusive Gateway** MAY use a marker that is shaped like an “X” and is placed within the **Gateway** diamond (see Figure 10.106) to distinguish it from other **Gateways**. This marker is NOT REQUIRED (see Figure 10.105).
- A diagram SHOULD be consistent in the use of the “X” internal indicator. That is, a diagram SHOULD NOT have some **Gateways** with an indicator and other **Gateways** without an indicator.

![Figure 10.105 - An Exclusive Data-Based Decision (Gateway) Example without the Internal Indicator](image-url)
A default path can optionally be identified, to be taken in the event that none of the conditional Expressions evaluate to true. If a default path is not specified and the Process is executed such that none of the conditional Expressions evaluates to true, a runtime exception occurs.

A converging Exclusive Gateway is used to merge alternative paths. Each incoming Sequence Flow token is routed to the outgoing Sequence Flow without synchronization.
The **Exclusive Gateway** element inherits the attributes and model associations of **Gateway** (see Table 8.46). Table 10.123 presents the additional attributes and model associations of the **Exclusive Gateway** element.

### Table 10.123 – ExclusiveGateway Attributes & Model Associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>default</strong>: SequenceFlow [0..1]</td>
<td>The Sequence Flow that will receive a token when none of the conditionExpressions on other outgoing Sequence Flows evaluate to true. The default Sequence Flow should not have a conditionExpression. Any such Expression SHALL be ignored.</td>
</tr>
</tbody>
</table>

### 10.5.3 Inclusive Gateway

A diverging **Inclusive Gateway** (Inclusive Decision) can be used to create alternative but also parallel paths within a Process flow. Unlike the **Exclusive Gateway**, all condition Expressions are evaluated. The true evaluation of one condition Expression does not exclude the evaluation of other condition Expressions. All Sequence Flows with a true evaluation will be traversed by a token. Since each path is considered to be independent, all combinations of the paths MAY be taken, from zero to all. However, it should be designed so that at least one path is taken.

- The **Inclusive Gateway** MUST use a marker that is in the shape of a circle or an “O” and is placed within the Gateway diamond (see Figure 10.108) to distinguish it from other Gateways.

![Figure 10.108 - An example using an Inclusive Gateway](image)

A default path can optionally be identified, to be taken in the event that none of the conditional Expressions evaluate to true. If a default path is not specified and the Process is executed such that none of the conditional Expressions evaluates to true, a runtime exception occurs.

A converging **Inclusive Gateway** is used to merge a combination of alternative and parallel paths. A control flow token arriving at an Inclusive Gateway MAY be synchronized with some other tokens that arrive later at this Gateway. The precise synchronization behavior of the Inclusive Gateway can be found on page 292.
The Inclusive Gateway element inherits the attributes and model associations of Gateway (see Table 8.46). Table 10.124 presents the additional attributes and model associations of the Inclusive Gateway element.

### Table 10.124 – InclusiveGateway Attributes & Model Associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>default</code>: SequenceFlow [0..1]</td>
<td>The Sequence Flow that will receive a token when none of the conditionExpressions on other Sequence Flows evaluate to true. The default Sequence Flow should not have a conditionExpression. Any such Expression SHALL be ignored.</td>
</tr>
</tbody>
</table>

#### 10.5.4 Parallel Gateway

A Parallel Gateway is used to synchronize (combine) parallel flows and to create parallel flows.

- The Parallel Gateway MUST use a marker that is in the shape of a plus sign and is placed within the Gateway diamond (see Figure 10.110) to distinguish it from other Gateways.
**Parallel Gateways** are used for synchronizing parallel flow (see Figure 10.111).

A **Parallel Gateway** creates parallel paths without checking any conditions; each *outgoing Sequence Flow* receives a *token* upon execution of this **Gateway**. For *incoming* flows, the **Parallel Gateway** will wait for all *incoming* flows before triggering the flow through its *outgoing Sequence Flows*. 
10.5.5 Complex Gateway

The Complex Gateway can be used to model complex synchronization behavior. An Expression \( activationCondition \) is used to describe the precise behavior. For example, this Expression could specify that \( tokens \) on three out of five \( incoming Sequence Flows \) are needed to activate the Gateway. What \( tokens \) are produced by the Gateway is determined by conditions on the \( outgoing Sequence Flows \) as in the split behavior of the Inclusive Gateway. If \( tokens \) arrive later on the two remaining \( Sequence Flows \), those \( tokens \) cause a reset of the Gateway and new \( token \) can be produced on the \( outgoing Sequence Flows \). To determine whether it needs to wait for additional \( tokens \) before it can reset, the Gateway uses the synchronization semantics of the Inclusive Gateway.

- The Complex Gateway MUST use a marker that is in the shape of an asterisk and is placed within the Gateway diamond (see Figure 10.113) to distinguish it from other Gateways.

Figure 10.112 - Parallel Gateway class diagram

Figure 10.113 - An example using a Complex Gateway
The **Complex Gateway** has, in contrast to other **Gateways**, an internal state, which is represented by the boolean *instance* attribute `waitingForStart`, which is initially *true* and becomes *false* after activation. This attribute can be used in the conditions of the *outgoing Sequence Flows* to specify where *tokens* are produced upon activation and where *tokens* are produced upon reset. It is RECOMMENDED that each *outgoing Sequence Flow* either get a *token* upon activation or upon reset but not both. At least one *outgoing Sequence Flow* should receive a *token* upon activation but a *token* MUST NOT be produced upon reset.

Figure 10.114 shows the class diagram for the **Complex Gateway**.

![Complex Gateway class diagram](image)

Figure 10.114 - Complex Gateway class diagram

The **Complex Gateway** element inherits the attributes and model associations of **Gateway** (see Table 8.46). Table 10.125 presents the additional model associations of the **Complex Gateway** element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>activationCondition</code>: Expression [0..1]</td>
<td>Determines which combination of <em>incoming tokens</em> will be synchronized for activation of the <strong>Gateway</strong>.</td>
</tr>
<tr>
<td><code>default</code>: SequenceFlow [0..1]</td>
<td>The <strong>Sequence Flow</strong> that will receive a <em>token</em> when none of the <em>conditionExpressions</em> on other <strong>Sequence Flows</strong> evaluate to <em>true</em>. The <strong>default Sequence Flow</strong> should not have a <em>conditionExpression</em>. Any such Expression SHALL be ignored.</td>
</tr>
</tbody>
</table>
10.5.6 Event-Based Gateway

The Event-Based Gateway represents a branching point in the Process where the alternative paths that follow the Gateway are based on Events that occur, rather than the evaluation of Expressions using Process data (as with an Exclusive or Inclusive Gateway). A specific Event, usually the receipt of a Message, determines the path that will be taken. Basically, the decision is made by another Participant, based on data that is not visible to Process, thus, requiring the use of the Event-Based Gateway.

For example, if a company is waiting for a response from a customer they will perform one set of Activities if the customer responds “Yes” and another set of Activities if the customer responds “No.” The customer’s response determines which path is taken. The identity of the Message determines which path is taken. That is, the “Yes” Message and the “No” Message are different Messages—i.e., they are not the same Message with different values within a property of the Message. The receipt of the Message can be modeled with an Intermediate Event with a Message trigger or a Receive Task. In addition to Messages, other triggers for Intermediate Events can be used, such as Timers.

The Event Gateway shares the same basic shape of the Gateways, a diamond, with a marker placed within the diamond to indicate variations of the Gateway.

- An Event Gateway is a diamond that MUST be drawn with a single thin line.
- The use of text, color, size, and lines for an Event Gateway MUST follow the rules defined in “Use of Text, Color, Size, and Lines in a Diagram” on page 41.
- The marker for the Event Gateway MUST look like a catch Multiple Intermediate Event (see Figure 10.115).

Figure 10.115 – Event-Based Gateway

Unlike other Gateways, the behavior of the Event Gateway is determined by a configuration of elements, rather than the single Gateway.

- An Event Gateway MUST have two or more outgoing Sequence Flows.
- The outgoing Sequence Flows of the Event Gateway MUST NOT have a conditionExpression.

The objects that are on the target end of the Gateway’s outgoing Sequence Flows are part of the configuration of the Gateway.
**Event-Based Gateways** are configured by having *outgoing Sequence Flows* target an *Intermediate Event* or a *Receive Task* in any combination (see Figure 10.116 and Figure 10.117) except that:

- If *Message Intermediate Events* are used in the configuration, then *Receive Tasks* MUST NOT be used in that configuration and vice versa.
- *Receive Tasks* used in an *Event Gateway* configuration MUST NOT have any attached *Intermediate Events*.
- Only the following *Intermediate Event triggers* are valid: Message, Signal, Timer, Conditional, and Multiple (which can only include the previous triggers). Thus, the following *Intermediate Event triggers* are not valid: Error, Cancel, Compensation, and Link.
- Target elements in an *Event Gateway* configuration MUST NOT have any additional *incoming Sequence Flows* (other than that from the Event Gateway).

![Figure 10.116 - An Event-Based Gateway example using Message Intermediate Events](image1)

![Figure 10.117 - An Event-Based Gateway example using Receive Tasks](image2)
When the first **Event** in the **Event Gateway** configuration is triggered, then the path that follows that **Event** will used (a *token* will be sent down the **Event**’s outgoing **Sequence Flows**). All the remaining paths of the **Event Gateway** configuration will no longer be valid. Basically, the **Event Gateway** configuration is a race condition where the first **Event** that is triggered wins.

There are variations of the **Event Gateway** that can be used at the start of the **Process**. The behavior and marker of the **Gateway** will change.

**Event Gateways** can be used to instantiate a **Process**. By default the **Gateway**’s `instantiate` attribute is `false`, but if set to `true`, then the **Process** is instantiated when the first **Event** of the **Gateway**’s configuration is triggered.

- If the **Event Gateway**’s `instantiate` attribute is set to `true`, then the marker for the **Event Gateway** looks like a **Multiple Start Event** (see Figure 10.118).

![Figure 10.118 – Exclusive Event-Based Gateway to start a Process](image)

In order for an **Event Gateway** to instantiate a **Process**, it MUST not have any *incoming* **Sequence Flows**.

In some situations a modeler might want the **Process** to be instantiated by one of a set of **Messages** while still requiring all of the **Messages** for the working of the same **Process instance**. To handle this, there is another variation of the **Event Gateway**.

- If the **Event Gateway**’s `instantiate` attribute is set to `true` and the `eventGatewayType` attribute is set to `Parallel`, then the marker for the **Event Gateway** looks like a **Parallel Multiple Start Event** (see Figure 10.119).

- The **Event Gateway**’s `instantiate` attribute MUST be set to `true` in order for the `eventGatewayType` attribute to be set to `Parallel` (i.e., for **Event Gateway**’s that do not instantiate the **Process** MUST be **Exclusive**—a standard **Parallel Gateway** can be used to include parallel **Events** in the middle of a **Process**).

![Figure 10.119 – Parallel Event-Based Gateway to start a Process](image)

The **Parallel Event Gateway** is also a type of race condition. In this case, however, when the first **Event** is triggered and the **Process** is instantiated, the other **Events** of the **Gateway** configuration are not disabled. The other **Events** are still waiting and are expected to be triggered before the **Process** can (normally) complete. In this case, the **Messages** that trigger the **Events** of the **Gateway** configuration MUST share the same correlation information.
The **Event-Based Gateway** element inherits the attributes and model associations of **Gateway** (see Table 8.46). Table 10.127 presents the additional attributes and model associations of the **Event-Based Gateway** element.

### Table 10.127 – EventBasedGateway Attributes & Model Associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>instantiate: boolean = false</td>
<td>When true, receipt of one of the Events will instantiate the Process instance.</td>
</tr>
</tbody>
</table>

The `eventGatewayType` determines the behavior of the Gateway when used to instantiate a Process (as described above). The attribute can only be set to parallel when the `instantiate` attribute is set to `true`.

**Event-Based Gateways** can be used at the start of a Process, without having to be a target of Sequence Flows. There can be multiple such **Event-Based Gateways** at the start of a Process. Ordinary Start Events and Event-Based Gateways can be used together.
10.5.7 Gateway Package XML Schemas

Table 10.128 – ComplexGateway XML schema

```xml
<xsd:element name="complexGateway" type="tComplexGateway" substitutionGroup="flowElement"/>
<xsd:complexType name="tComplexGateway">
  <xsd:complexContent>
    <xsd:extension base="tGateway">
      <xsd:sequence>
        <xsd:element name="activationCondition" type="tExpression" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="default" type="xsd:IDREF"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 10.129 – EventBasedGateway XML schema

```xml
<xsd:element name="eventBasedGateway" type="tEventBasedGateway" substitutionGroup="flowElement"/>
<xsd:complexType name="tEventBasedGateway">
  <xsd:complexContent>
    <xsd:extension base="tGateway">
      <xsd:attribute name="instantiate" type="xsd:boolean" default="false"/>
      <xsd:attribute name="eventGatewayType" type="tEventBasedGatewayType" default="Exclusive"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

```xml
<xsd:simpleType name="tEventBasedGatewayType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Exclusive"/>
    <xsd:enumeration value="Parallel"/>
  </xsd:restriction>
</xsd:simpleType>
```

Table 10.130 – ExclusiveGateway XML schema

```xml
<xsd:element name="exclusiveGateway" type="tExclusiveGateway" substitutionGroup="flowElement"/>
<xsd:complexType name="tExclusiveGateway">
  <xsd:complexContent>
    <xsd:extension base="tGateway">
      <xsd:attribute name="default" type="xsd:IDREF" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 10.131 – Gateway XML schema

```xml
<xsd:element name="gateway" type="tGateway" abstract="true"/>
<xsd:complexType name="tGateway">
```
10.6 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 performs the steps necessary to reverse the effects of an Activity. In case of a Sub-Process, the compensation handler 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.
10.6.1 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, which is connected to the boundary Event through an Association (see Figure 10.121). The Compensation Activity, which can be either a Task or a Sub-Process, has a marker to show that it is used for compensation only and is outside the normal flow of the Process.

Figure 10.121 - Compensation through a boundary Event

A Compensation Event Sub-Process is contained within a Process or a Sub-Process (see Figure 10.122). Like the Compensation Activity, the Compensation Event Sub-Process is outside the normal flow of the Process. The Event Sub-Process, which is marked with a dotted line boundary, can access data that is part of its parent, a snapshot at the point in time when its parent completed. A Compensation Event Sub-Process can recursively trigger compensation for Activities contained in 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**.

The example in Figure 10.122, above contains a custom **Compensation Event Sub-Process**, triggered by a **Compensation Start Event**. Note that this **compensation handler** deviates from default compensation in that it runs **Compensation Activities** in an order different from the order in the forward case; it also contains an additional **Activity** adding **Process** logic that cannot be derived from the body of the **Sub-Process** itself.

### 10.6.2 Compensation Triggering

**Compensation** is triggered using a **compensation throw 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 compensation throw Event waits for the completion of the triggered compensation handler. Alternatively, compensation can just be triggered without waiting for its completion, by setting the `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.

10.6.3 Relationship between Error Handling and Compensation

The following items define the relationship between error handling and compensation:

- Compensation employs a “presumed abort principle,” with the following consequences: Compensation of a failed Activity results in a null operation.
- When an Activity fails, i.e., is left because an error has been thrown, it’s the error handlers responsibility to ensure that no further compensation will be necessary once the error handler has completed.
- 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 is thrown, ensuring the behavior for auditing and monitoring.

10.7 Lanes

A Lane is a sub-partition within a Process (often within a Pool) and will extend the entire length of the Process level, either vertically (see Figure 10.122) or horizontally (see Figure 10.123). Text associated with the Lane (e.g., its name and/or that of any Process element attribute) can be placed inside the shape, in any direction or location, depending on the preference of the modeler or modeling tool vendor. Our examples place the name as a banner on the left side (for horizontal Pools) or at the top (for vertical Pools) on the other side of the line that separates the Pool name, however, this is not a requirement.

- A Lane is a square-cornered rectangle that MUST be drawn with a solid single line (see Figure 10.123 and Figure 10.124).
- The label for the Lane MAY be placed in any location and direction within the Lane, but MUST NOT be separated from the contents of the Lane by a single line (except in the case that there are sub-Lanes within the Lane).
Lanes are used to organize and categorize Activities within a Pool. The meaning of the Lanes is up to the modeler. BPMN does not specify the usage of Lanes. Lanes are often used for such things as internal roles (e.g., Manager, Associate), systems (e.g., an enterprise application), an internal department (e.g., shipping, finance), etc. In addition, Lanes can be nested (see Figure 10.125) or defined in a matrix. For example, there could be an outer set of Lanes for company departments and then an inner set of Lanes for roles within each department.
Figure 10.125 - An Example of Nested Lanes

Figure 10.126 shows the Lane class diagram. When a Lane is defined it is contained within a LaneSet, which is contained within a Process.
The LaneSet element defines the container for one or more Lanes. A Process can contain one or more LaneSets. Each LaneSet and its Lanes can partition the Flow Nodes in a different way.

The LaneSet element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 10.134 presents the additional attributes and model associations of the LaneSet element.

**Table 10.134 – LaneSet attributes and model associations**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string [0..1]</td>
<td>The name of the LaneSet. A LaneSet is not visually displayed on a BPMN diagram. Consequently, the name of the LaneSet is not displayed as well.</td>
</tr>
<tr>
<td>process: Process</td>
<td>The Process owning the LaneSet</td>
</tr>
<tr>
<td>lanes: Lane [0..*]</td>
<td>One or more Lane elements, which define a specific partition in the LaneSet.</td>
</tr>
<tr>
<td>parentLane: Lane [0..1]</td>
<td>The reference to a Lane element which is the parent of this LaneSet.</td>
</tr>
</tbody>
</table>

A Lane element defines one specific partition in a LaneSet. The Lane can define a partition element that specifies the value and element type, a tool can use to determine the list of Flow Nodes to be partitioned into this Lane. All Lanes in a single LaneSet MUST define partition element of the same type, e.g., all Lanes in a LaneSet reference a Resource as the partition element, but each Lane references a different Resource instance.

The Lane element inherits the attributes and model associations of BaseElement (see Table 8.5). Table 10.135 presents the additional attributes and model associations of the Lane element.
10.8 Process Instances, Unmodeled Activities, and Public Processes

A Process can be executed or performed many times, but each time is expected to follow the steps laid out in the Process model. For example, the Process in Figure 10.1 will occur every Friday, but each instance is expected to perform Task “Receive Issue List,” then Task “Review Issue List,” and so on, as specified in the model. Each instance of a Process is expected to be valid for the model, but some instances might not, for example if the Process has manual Activities, and the performers have not had proper instruction on how to carry out the Process.

In some applications it is useful to allow more Activities and Events to occur when a Process is executed or performed than are contained in the Process model. This enables other steps to be taken as needed without changing the Process. For example, instances of the Process in Figure 10.1 might execute or perform an extra Activity between Task “Receive Issue List” and Task “Review Issue List.” These instances are still valid for the Process model in Figure 10.1, because the instances still execute or perform the Activities in the Process, in the order they are modeled and under conditions specified for them.

There are two ways to specify whether unmodeled Activities are allowed to occur in Process instances:

- If the isClosed attribute of a Process has a value of false or no value, then interactions, such as sending and receiving Messages and Events, MAY occur in an instance without additional flow elements in the Process. Unmodeled interactions can still be restricted on particular Sequence Flow in the Process (see next bullet). If the isClosed attribute of a Process has a value of true, then interactions, such as sending and receiving Messages and Events, MAY NOT occur without additional flow elements in the Process. This restriction overrides any unmodeled interactions allowed by Sequence Flows in the next bullet.

- If the isImmediate attribute of a Sequence Flow in a Process has a value of false, then other Activities and interactions not modeled in the Process MAY be executed or performed during the Sequence Flow. If the isImmediate attribute has a value of true, then Activities and interactions not modeled in the Process MAY NOT be executed or performed during Sequence Flow. In non-executable Processes (isExecutable attribute has value false, or defaults to false), Sequence Flows with no value for isImmediate are treated as if the

---

Table 10.135 – Lane attributes and model associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: string</td>
<td>The name of the Lane</td>
</tr>
<tr>
<td>partitionElement:</td>
<td>A reference to a BaseElement that specifies the partition value and partition</td>
</tr>
<tr>
<td></td>
<td>type. Using this partition element a BPMN compliant tool can determine the</td>
</tr>
<tr>
<td></td>
<td>FlowElements that have to be partitioned in this Lane.</td>
</tr>
<tr>
<td>partitionElementRef:</td>
<td>A reference to a BaseElement that specifies the partition value and partition</td>
</tr>
<tr>
<td></td>
<td>type. Using this partition element a BPMN compliant tool can determine the</td>
</tr>
<tr>
<td></td>
<td>FlowElements that have to be partitioned in this Lane.</td>
</tr>
<tr>
<td>childLaneSet:</td>
<td>A reference to a LaneSet element for embedded Lanes.</td>
</tr>
<tr>
<td>flowNodeRefs:</td>
<td>The list of FlowNodes partitioned into this Lane according to the partitionElement defined as part of the Lane element.</td>
</tr>
</tbody>
</table>
value were false. In executable Processes (isExecutable attribute has value true, or defaults to true), Sequence Flows with no value for isImmediate are treated as if the value were true. Executable Processes cannot have a false value for the isImmediate attribute.

Restrictions on unmodeled Activities specified with isClosed and isImmediate apply only under executions or performances (instances) of the Process containing the restriction. These Activities MAY occur in instances of other Processes.

When a Process allows Activities to occur that the Process does not model, those Activities might appear in other Process models. The executions or performances (instances) of these other Processes might be valid for the original Process. For example, a Process might be defined similar to the one in Figure 10.1 that adds an extra Activity between Task “Receive Issue List” and Task “Review Issue List.” The Process in Figure 10.1 might use isClosed or isImmediate to allow other Activities to occur in between Task “Receive Issue List” and Task “Review Issue List.” When the Process is executed or performed, then instances of the other Process (the one with the extra step in between Task “Receive Issue List” and Task “Review Issue List”) will be valid for the Process in Figure 10.1.

Modelers can declare that they intend all instances of one Process will be valid for another Process using the supports association between the Processes. During development of these Processes, support might not actually hold, because the association just expresses modeler intent.

A common use for model support is between private and public Processes, see “Overview” (page 22). A public Process contains Activities visible to external parties, such as Participants in a Collaboration, while a private Process includes other Activities that are not visible to external parties. The hidden Activities in a private Process are not modeled in the public Process. However, it is expected that instances of the private Process will appear to external parties as if they could be instances of the public Process. This means the private Process supports the public Process (it is expected that all instances of the private Process will be valid for the public one).

A Process that supports another, as a private Process can to a public Process, does not need to be entirely similar to the other Process. It is only REQUIRED that instances of the Process appear as if they could be instance of the other Process. For example Figure 10.127 shows a public Process at the top with a Send Task and Receive Task. A supporting private Process is shown at the bottom. The private Process sends and receives the same Messages, but using Events instead of Tasks. It also introduces Activities not modeled in the public Process. However all instances of the private Process will appear as if they could be instances of the public one, because the Messages are sent and received in the order REQUIRED by the public Process, and the public Process allows unmodeled Activities to occur.

Figure 10.127 - One Process supporting to another
In practice, a public Process looks like an underspecified private Process. Anything not specified in the public Process is determined by the private one. For example, if none of the outgoing Sequence Flows for an Exclusive Gateway have conditionExpressions, the private Process will determine which one of the Activities targeted by the Sequence Flows will occur. Another example is a Timer Event with no EventDefinition. The private Process will determine when the timer goes off.

10.9 Auditing

The Auditing element and its model associations allow defining attributes related to auditing. It leverages the BPMN extensibility mechanism. This element is used by FlowElements and Process. The actual definition of auditing attributes is out of scope of this specification. BPMN 2.0 implementations can define their own set of attributes and their intended semantics.

10.10 Monitoring

The Monitoring and its model associations allow defining attributes related to monitoring. It leverages the BPMN extensibility mechanism. This element is used by FlowElements and Process. The actual definition of monitoring attributes is out of scope of this specification. BPMN 2.0 implementations can define their own set of attributes and their intended semantics.
10.11 Process Package XML Schemas

Table 10.136 – Process XML schema

```
<xsd:element name="process" type="tProcess" substitutionGroup="rootElement"/>
<xsd:complexType name="tProcess">
    <xsd:complexContent>
        <xsd:extension base="tCallableElement">
            <xsd:sequence>
                <xsd:element ref="auditing" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="monitoring" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="processRole" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="property" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="laneSet" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="flowElement" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="artifact" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="resourceRole" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="correlationSubscription" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="supports" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="processType" type="tProcessType" default="None"/>
            <xsd:attribute name="isExecutable" type="xsd:boolean" use="optional"/>
            <xsd:attribute name="isClosed" type="xsd:boolean" default="false"/>
            <xsd:attribute name="definitionalCollaborationRef" type="xsd:QName" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Figure 10.129 - Monitoring Class Diagram
Table 10.137 – Auditing XML schema

```xml
<xsd:element name="auditing" type="tAuditing"/>
<xsd:complexType name="tAuditing">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.138 – GlobalTask XML schema

```xml
<xsd:element name="globalTask" type="tGlobalTask" substitutionGroup="rootElement"/>
<xsd:complexType name="tGlobalTask">
    <xsd:complexContent>
        <xsd:extension base="tCallableElement">
            <xsd:sequence>
                <xsd:element ref="resourceRole" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.139 – Lane XML schema

```xml
<xsd:element name="lane" type="tLane"/>
<xsd:complexType name="tLane">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="partitionElement" type="tBaseElement" minOccurs="0" maxOccurs="1"/>
                <xsd:element name="flowNodeRef" type="xsd:IDREF" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="childLaneSet" type="tLaneSet" minOccurs="0" maxOccurs="1"/>
                <xsd:attribute name="name" type="xsd:string"/>
                <xsd:attribute name="partitionElementRef" type="xsd:QName"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10.140 – LaneSet XML schema

```xml
<xsd:element name="laneSet" type="tLaneSet"/>
<xsd:complexType name="tLaneSet">
</xsd:complexType>
```
<xsd:extension base="tBaseElement">
  <xsd:sequence>
    <xsd:element ref="lane" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

**Table 10.141 – Monitoring XML schema**

<xsd:element name="monitoring" type="tMonitoring"/>
<xsd:complexType name="tMonitoring">
  <xsd:complexContent>
    <xsd:extension base="tBaseElement"/>
  </xsd:complexContent>
</xsd:complexType>

**Table 10.142 – Performer XML schema**

<xsd:element name="performer" type="tPerformer" substitutionGroup="resourceRole"/>
<xsd:complexType name="tPerformer">
  <xsd:complexContent>
    <xsd:extension base="tResourceRole"/>
  </xsd:complexContent>
</xsd:complexType>
11 Choreography

Note – The content of this chapter is REQUIRED for BPMN Choreography Modeling Conformance or for BPMN Complete Conformance. However, this chapter is NOT REQUIRED for BPMN Process Modeling Conformance, BPMN Process Execution Conformance, or BPMN BPEL Process Execution Conformance. For more information about BPMN conformance types, see page 2.

A Choreography is a type of process, but differs in purpose and behavior from a standard BPMN Process. A standard Process, or an Orchestration Process (see page 145), is more familiar to most process modelers and defines the flow of Activities of a specific PartnerEntity or organization. In contrast, Choreography formalizes the way business Participants coordinate their interactions. The focus is not on orchestrations of the work performed within these Participants, but rather on the exchange of information (Messages) between these Participants.

Another way to look at Choreography is to view it as a type of business contract between two or more organizations. This entails Message (document) exchanges in an orderly fashion: e.g., first a retailer sends a purchase order request to a supplier; next the supplier either confirms or rejects intention to investigate the order; then supplier proceeds to investigate stock for line-items and seeks outside suppliers if necessary; accordingly the supplier sends a confirmation or rejection back; during this period the retailer can send requests to vary the order, etc.

Message exchanges between partners go beyond simple request-response interactions into multi-cast, contingent requests, competing receives, streaming, and other service interaction patterns (REF for SIP). Moreover, they cluster around distinct scenarios such as: creation of sales orders; assignment of carriers of shipments involving different sales orders; managing the “red tape” of crossing customs and quarantine; processing payment and investigating exceptions. A Choreography is a definition of expected behavior, basically a procedural business contract, between interacting Participants (see page 112 for more information on Participants). It brings Message exchanges and their logical relation as Conversations into view. This allows partners to plan their Business Processes for inter-operation without introducing conflicts. An example of a conflict could arise if a retailer was allowed to send a variation on a purchase order immediately after sending the initial request. The Message exchange sequences in Choreography models need to be reflected in the orchestration Processes of participants. A Choreography model makes it possible to derive the Process interfaces of each partner’s Process (REF: Decker & Weske, 2007).

To leverage the familiarity of flow charting types of Process models, BPMN Choreographies also have “activities” that are ordered by Sequence Flows. These “activities” consist of one or more interactions between Participants. These interactions are often described as being message exchange patterns (MEPs). A MEP is the atomic unit (“Activity”) of a Choreography.

Some MEPs involve a single Message (e.g., a “Customer” requests an “Order” from a “Supplier”). Other MEPs will involve two Messages in a request and response format (e.g., a “Supplier” request a “Credit Rating” from a “Financial Institution,” who then returns the “Credit Rating” to the “Supplier”). There can be even more complex MEPs that involve error Messages, for example.

A single MEP can be defined as a BPMN Choreography Task (see page 323). Thus, a Choreography defines the order in which Choreography Tasks occur. Sub-Choreographies allow the composition/decomposition of Choreographies.

Choreographies are designed in BPMN to allow stand-alone, scalable models of these Participant interactions. However, since BPMN provides other Business Process modeling views, Choreographies are designed to fit within BPMN Collaboration diagrams to display the relationship between the Choreography and Orchestration Processes (thus, expanding BPMN 1.2 capabilities—see page 109, above, for more information on Collaborations, and page 362 for Choreographies within Collaborations).
Figure 11.1 displays the metamodel of the key BPMN elements that contribute to Choreography modeling. The sections of this chapter will describe the characteristics of these elements and how they are used in a Choreography.

The Choreography element inherits the attributes and model associations of Collaboration (see Table 9.1) and of FlowElementContainer (see Table 8.45), but does not have any additional attributes or model associations.

Note – The Collaboration attribute choreographyRef is not applicable to Choreography.

11.1 Basic Choreography Concepts

A key to understanding Choreographies and how they are used in BPMN is their relationship to Pools (see page 112 for more information on Pools). Choreographies exist outside of or in between Pools. A Process, within a Pool, represents the work of a specific PartnerEntity (e.g., “FedEx”), often substituted by a PartnerRole (e.g., “Shipper”) when a PartnerEntity is not identified and can be decided later. The PartnerEntity/PartnerRole is called a Participant in BPMN. Pools are the graphical representation of Participants. A Choreography, on the other hand, is a different kind of process. A Choreography defines the sequence of interactions between Participants. Thus,
a **Choreography** does not exist in a single **Pool**—it is not the purview of a single **Participant**. Each step in the **Choreography** involves two or more **Participants** (these steps are called **Choreography Activities**—see below). This means that the **Choreography**, in BPMN terms, is defined outside of any particular **Pool**.

The key question that needs to be continually asked during the development of a **Choreography** is “what information do the **Participants** in the **Choreography** have?” Basically, each **Participant** can only understand the status of the **Choreography** through observable behavior of the other **Participants**—which are the **Messages** that have been sent and received. If there are only two **Participants** in the **Choreography**, then it is very simple—both **Participants** will be aware of who is responsible for sending the next **Message**. However, if there are more than two **Participants**, then the modeler needs to be careful to sequence the **Choreography Activities** in such a way that the **Participants** know when they are responsible for initiating the **interactions**.

Figure 11.2 presents a sample **Choreography**. The details of **Choreography** behavior and elements will be described in the sections below.

![Figure 11.2 - An example of a Choreography](image)

To illustrate the correspondence between **Collaboration** and **Choreography**, consider an example from logistics. Figure 11.3 shows a **Collaboration** where the **Pools** are expanded to reveal orchestration details per participant (for **Shipper**, **Retailer** etc.). **Message Flows** connect the elements in the different **Pools** related to different participants, indicating **Message** exchanges. For example, a **Planned Order Variations Message** is sent by the **Supplier** to the **Retailer**; the corresponding send and receive have been modeled using regular BPMN messaging **Activities**. Also, a number of **Messages** of the same type being sent, for example a number of **Retailer Order and Delivery Variations Messages** can be sent from the **Retailer** to the **Supplier**, indicated by respective **multi-instances** constructs (for brevity, the actual elements for sending/receiving inside the **multi-instances** construct have been omitted).
The scenario modeled in Figure 11.3 entails shipment planning for the next supply replenishment variations: the Supplier confirms all previously accepted variations for delivery with the Retailer; the Retailer sends back a number of further possible variations; the Supplier requests to the Shipper and Consignee possible changes in delivery; accordingly, the Retailer interacts with the Consignee and Supplier for final confirmations.

A problem with model interconnections for complex Choreographies is that they are vulnerable to errors – interconnections might not be sequenced correctly, since the logic of Message exchanges is considered from each partner at a time. This in turn leads to deadlocks. For example, consider the PartnerRole of Retailer in Figure 11.3 and assume that, by error, the order of Confirmation Delivery Schedule and Retailer Confirmation received (far right) were swapped. This would result in a deadlock since both, Retailer and Consignee would wait for the other to send a Message. Deadlocks in general, however, are not that obvious and might be difficult to recognize in a Collaboration.

Figure 11.4 shows the Choreography corresponding to the Collaboration of Figure 11.3.
11.2 Data

A Choreography does not have a central control mechanism and there is no mechanism for maintaining any central Process (Choreography) data. Thus, any element in a Process that would normally depend on conditional or assignment expressions, would not have any central source for this data to be maintained and understood by all the Participants involved in the Choreography.

As mentioned above, neither Data Objects nor Repositories are used in Choreographies. Both of these elements are used exclusively in Processes and require the concept of a central locus of control. Data Objects are basically variables and there would be no central system to manage them. Data can be used in expressions that are used in Exclusive Gateways, but only that data which has been sent through a Message in the Choreography.

11.3 Use of BPMN Common Elements

Some BPMN elements are common to both Process and Choreography diagrams, as well as Collaboration; they are used in these diagrams. The next few sections will describe the use of Messages, Message Flows, Participants, Sequence Flows, Artifacts, Correlations, Expressions, and Services in Choreography.

The key graphical elements of Gateways and Events are also common to both Choreography and Process. Since their usage has a large impact, they are described in major sections of this chapter (see page 339 for Events and page 344 for Gateways).
11.3.1 Sequence Flow

Sequence Flows are used within Choreographies to show the sequence of the Choreography Activities, which can have intervening Gateways. They are used in the same way as they are in Processes. They are only allowed to connect with other Flow Objects. For Processes, they can only connect Events, Gateways, and Activities. For Choreographies, they can only connect Events, Gateways, and Choreography Activities (see Figure 11.5).

![Sequence Flow in a Choreography](image)

Figure 11.5 - The use of Sequence Flows in a Choreography

There are two additional variations of Sequence Flows:

- **Conditional Sequence Flows**: Conditions can be added to Sequence Flows in two situations:
  - From Gateways: Outgoing Sequence Flows have conditions for Exclusive and Inclusive Gateways. The data referenced in the conditions need to be visible to two or more Participants in the Choreography. The data becomes visible if it is part of a Message that had been sent (previously) within the Choreography. See page 345 and page 352 for more information about how Exclusive and Inclusive Gateways are used in Choreography.
  - From Choreography Activities: Outgoing Sequence Flows may have conditions for Choreography Activities. Since these act similar to Inclusive Gateways, the Conditional Sequence Flows can be used in Choreographies. The conditions have the same restrictions that apply to the visibility of the data for Gateways.

- **Default Sequence Flows**: For Exclusive Gateways, Inclusive Gateways, and Choreography Activities that have Conditional Sequence Flows, one of the outgoing Sequence Flows may be a Default Sequence Flow. Because the other outgoing Sequence Flows will have appropriately visible data as described above, the Participants would know if all the other conditions would be false, thus the Default Sequence Flow would be selected and the Choreography would move down that Sequence Flow.

In some applications it is useful to allow additional Messages that are not part of the defined Choreography model to be sent between Participants when the Choreography is carried out. This enables Participants to exchange other Messages as needed without changing the Choreography. There are two ways to specify this:

- If the isClosed attribute (from Collaboration) of a Choreography has a value of false or no value, then Participants may send Messages to each other without additional Choreography Activities in the Choreography. Unmodeled messaging can be restricted on particular Sequence Flows in the Choreography, see next bullet. If the isClosed attribute of a Choreography has a value of true, then Participants may not send Messages to each other without additional Choreography Activities in the Choreography. This restriction overrides any unmodeled messaging allowed by Sequence Flows in the next bullet.
• If the `isImmediate` attribute of a `Sequence Flow` has a value of `false` or no value, then `Participants` MAY send `Messages` to each other between the elements connected by the `Sequence Flow` without additional `Choreography Activities` in the `Choreography`. If the `isImmediate` attribute of a `Sequence Flow` has a value of `true`, then `Participants` MAY NOT send `Messages` to each other between the elements connected by the `Sequence Flow` without additional `Choreography Activities` in the `Choreography`. The value of `isImmediate` attribute of a `Sequence Flow` has no effect if the `isClosed` attribute of the containing `Choreography` has a value of `true`.

Restrictions on unmodeled messaging specified with `isClosed` and `isImmediate` applies only under the `Choreography` containing the restriction. `PartnerEntities` and `PartnerRoles` of the `Participants` MAY send `Messages` to each other under other `Choreographies`, `Collaborations`, and `Conversations`.

11.3.2 Artifacts

Both `Text Annotations` and `Groups` can be used within `Choreographies` and all `BPMN` diagrams. There are no restrictions on their use.

11.4 Choreography Activities

A `Choreography Activity` represents a point in a `Choreography` flow where an `interaction` occurs between two or more `Participants`.

The `Choreography Activity` class is an abstract element, sub-classing from `FlowElement` (as shown in Figure 11.6). When `Choreography Activities` are defined they are contained within a `Choreography` or a `Sub-Choreography`, which are `FlowElementContainers` (other `FlowElementContainers` are not allowed to contain `Choreography Activities`).
The **Choreography Activity** element inherits the attributes and model associations of **FlowElement** (see Table 8.44) through its relationship to **FlowNode**. Table 11.1 presents the additional model associations of the **Choreography Activity** element.

### Table 11.1 – Choreography Activity Model Associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>participantRefs</strong>: Participant [2..*]</td>
<td>A <strong>Choreography Activity</strong> has two or more Participants (see page 114 for more information on Participants).</td>
</tr>
<tr>
<td><strong>initiatingParticipantRef</strong>: Participant</td>
<td>One of the Participants will be the one that initiates the <strong>Choreography Activity</strong>.</td>
</tr>
<tr>
<td><strong>loopType</strong>: ChoreographyLoopType = None</td>
<td>A <strong>Choreography Activity</strong> MAY be performed once or MAY be repeated. The <strong>loopType</strong> attribute will determine the appropriate marker for the <strong>Choreography Activity</strong> (see below).</td>
</tr>
<tr>
<td><strong>correlationKeys</strong>: CorrelationKey [0..*]</td>
<td>This association specifies <strong>correlationKeys</strong> used by the <strong>Message Flow</strong> in the <strong>Choreography Activity</strong>, including <strong>Sub-Choreographies</strong> and called <strong>Choreographies</strong>.</td>
</tr>
</tbody>
</table>
11.4.1 Choreography Task

A Choreography Task is an atomic Activity in a Choreography Process. It represents an Interaction, which is one or two Message exchanges between two Participants. Using a Collaboration diagram to view these elements (see page 109 for more information on Collaboration), we would see the two Pools representing the two Participants of the Interaction (see Figure 11.7). The communication between the Participants is shown as a Message Flow.

![Figure 11.7 - A Collaboration view of Choreography Task elements](image)

In a Choreography diagram, this Interaction is collapsed into a single object, a Choreography Task. The name of the Choreography Task and each of the Participants are all displayed in the different bands that make up the shape’s graphical notation. There are two or more Participant Bands and one Task Name Band (see Figure 11.8).

- The Participant Band of the Participant that does not initiate the interaction MUST be shaded with a light fill.

![Figure 11.8 - A Choreography Task](image)
The interaction defined by a **Choreography Task** can be shown in an expanded format through a **Collaboration diagram** (see Figure 11.9, above—see page 109 for more information on **Collaborations**). In the **Collaboration** view, the **Participants** of the **Choreography Task Participant Band**’s will be represented by **Pools**. The interaction between them will be a **Message Flow**.

**Figure 11.9 - A Collaboration view of a Choreography Task**

**Figure 11.10 - A two-way Choreography Task**
In a Choreography Diagram, the Choreography Task object shares the same shape as a Task or any other BPMN Activity, which is a rectangle that has rounded corners.

- A Choreography Task is a rounded corner rectangle that MUST be drawn with a single line.
- The use of text, color, size, and lines for a Choreography Task MUST follow the rules defined in “Use of Text, Color, Size, and Lines in a Diagram” on page 41.

The three bands in the Choreography Task shape provide the distinction between this type of Task and an Orchestration Task (in a traditional BPMN diagram).

The Message sent by either one or both of the Participants of the Choreography Task can be displayed (see Figure 11.10, above). The Message icon is shown tethered to the Participant that is the sender of the Message.

- If the Message is the initiating Message of the Choreography Task, then the Message icon MUST be unfilled.
- If the Message is a return Message for the Choreography Task, then the Message icon MUST have a light fill.

Note that Messages can be tethered to a Call Choreography that references a GlobalChoreographyTask, but cannot be used for Sub-Choreographies or Call Choreography that references another Choreography.

As with a standard Orchestration Task, the Choreography Task MAY have internal markers to show how the Choreography Task MAY be repeated. There are two types of internal markers (see Figure 11.12):

- A Choreography Task MAY have only one of the three markers at one time.
- The marker for a Choreography Task that is a standard loop MUST be a small line with an arrowhead that curls back upon itself. The loopType of the Choreography Task MUST be Standard.
The marker for a **Choreography Task** that is parallel *multi-instance* MUST be a set of three vertical lines. The **loopType** of the **Choreography Task** MUST be **MultiInstanceParallel**.

The marker for a **Choreography Task** that is sequential *multi-instance* MUST be a set of three horizontal lines. The **loopType** of the **Choreography Task** MUST be **MultiInstanceSequential**.

The marker that is present MUST be centered at the bottom of the **Task Name Band** of the shape.

---

**Figure 11.12 - Choreography Task Markers**

**Figure 11.13 - The Collaboration view of a looping Choreography Task**
There are situations when a Participant for a Choreography Task is actually a multi-instance Participant. A multi-instance Participant represents a situation where there are more than one possible related Participants (PartnerRoles/PartnerEntities) that might be involved in the Choreography. For example, in a Choreography that involves the shipping of a product, there can be more than one type of shipper used, depending on the destination. When a Participant in a Choreography contains multiple instances, then a multi-instance marker will be added to the Participant Band for that Participant (see Figure 11.15).

- The marker for a Choreography Task that is multi-instance MUST be a set of three vertical lines.
- The marker that is present MUST be centered at the bottom of the Participant Band of the shape.

The width of the Participant Band will be expanded to contain both the name of the Participant and the multi-instance marker.

---

**Figure 11.14 - The Collaboration view of a Parallel Multi-Instance Choreography Task**

**Figure 11.15 - A Choreography Task with a multiple Participant**
The Choreography Task element inherits the attributes and model associations of Choreography Activity (see Table 11.1). Table 11.2 presents the additional model associations of the Choreography Task element.

### Table 11.2 – Choreography Task Model Associations

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>messageFlowRef: Message Flow [1..*]</td>
<td>Although not graphical represented, Choreography Task contains one or more Message Flows that represent the interaction(s) between the Participants referenced by the Choreography Task.</td>
</tr>
</tbody>
</table>

### 11.4.2 Sub-Choreography

A Sub-Choreography is a compound Activity in that it has detail that is defined as a flow of other Activities, in this case, a Choreography. Each Sub-Choreography involves two or more Participants. The name of the Sub-Choreography and each of the Participants are all displayed in the different bands that make up the shape’s graphical notation. There are two or more Participant Bands and one Sub-Process Name Band.

The Sub-Choreography can be in a collapsed view that hides its details (see Figure 11.17) or a Sub-Choreography can be expanded to show its details (a Choreography Process) within the Choreography Process in which it is contained (see Figure 11.19). In the collapsed form, the Sub-Process object uses a marker to distinguish it as a Sub-Choreography, rather than a Choreography Task.

The Sub-Process marker MUST be a small square with a plus sign (+) inside. The square MUST be positioned at the bottom center of the Sub-Process Name Band within the shape.

* The Participant Band of the Participant that does not initiate the interaction MUST be shaded with a light fill.
Figure 11.17 - A Sub-Choreography

Figure 11.18 shows an example of a potential Collaboration view of the above Sub-Choreography.

Figure 11.18 - A Collaboration view of a Sub-Choreography
Figure 11.19 shows an example of an expanded Sub-Choreography.

Figure 11.19 - An expanded Sub-Choreography

Figure 11.20 shows an example of a potential Collaboration view of the above Sub-Choreography.

Figure 11.20 - A Collaboration view of an expanded Sub-Choreography
The *Parent* Sub-Choreography (Expanded)

The **Choreography Activity** shares the same shape as a **Sub-Process** or any other **BPMN Activity**, which is in this state.

- A **Sub-Choreography** is a rounded corner rectangle that **MUST** be drawn with a single thin line.
- The use of text, color, size, and lines for a **Sub-Choreography** **MUST** follow the rules defined in “Use of Text, Color, Size, and Lines in a Diagram” on page 41.

The three or more partitions in the **Sub-Choreography** shape provide the distinction between this type of **Task** and an Orchestration **Sub-Process** (in a traditional **BPMN** diagram).

It is possible for a **Sub-Choreography** to involve more than two **Participants**. In this case, an additional **Participant Band** will be added to the shape for each additional **Participant** (see Figure 11.21). The ordering and position of the **Participant Band** (either in the upper or lower positions) is up to the modeler or modeling tool. In addition, any **Participant Band** beyond the first two **optional**; it is displayed at the discretion of the modeler or modeling tool. However, each **Participant Band** that is added **MUST** be added to the upper and lower sections of the **Sub-Choreography** in an alternative manner.

![Figure 11.21 - Sub-Choreography (Collapsed) with More than Two Participants](image)

As with a standard Orchestration **Sub-Process**, the **Sub-Choreography** **MAY** have internal markers to show how the **Sub-Choreography** **MAY** be repeated. There are two types of internal markers (see Figure 11.22):

- A **Sub-Choreography** **MAY** have only one of the three markers at one time.
- The marker for a **Sub-Choreography** that is a standard **loop** **MUST** be a small line with an arrowhead that curls back upon itself. The **loopType** of the **Sub-Choreography** **MUST** be **Standard**.
- The marker for a **Sub-Choreography** that is parallel **multi-instance** **MUST** be a set of three vertical lines. The **loopType** of the **Sub-Choreography** **MUST** be **MultiInstanceParallel**.
- The marker for a **Sub-Choreography** that is sequential **multi-instance** **MUST** be a set of three horizontal lines. The **loopType** of the **Sub-Choreography** **MUST** be **MultiInstanceSequential**.
- The marker that is present **MUST** be centered at the bottom of the **Sub-Process Name Band** of the shape.
There are situations when a Participant for a Sub-Choreography is actually a multi-instance Participant. A multi-instance Participant represents a situation where there are more than one possible related Participants (PartnerRoles/PartnerEntities) that can be involved in the Choreography. For example, in a Choreography that involves the shipping of a product, there can be more than one type of shipper used, depending on the destination. When a Participant in a Choreography contains multiple instances, then a multi-instance marker will be added to the Participant Band for that Participant (see Figure 11.23).

- The marker for a Sub-Choreography that is multi-instance MUST be a set of three vertical lines.
- The marker that is present MUST be centered at the bottom of the Participant Band of the shape.
- The width of the Participant Band will be expanded to contain both the name of the Participant and the multi-instance marker.

This includes Compensation Event Sub-Processes (contained within a Sub-Choreography) as well as the external Compensation Activity connected through an Association.

The Sub-Choreography element inherits the attributes and model associations of Choreography Activity (see Table 11.1) and FlowElementsContainer (see Table 8.45). Table 11.3 presents the additional model associations of the GlobalChoreographyTask element:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>artifacts: Artifact [0..*]</td>
<td>This attribute provides the list of Artifacts that are contained within the Sub-Choreography.</td>
</tr>
</tbody>
</table>
11.4.3 Call Choreography

A Call Choreography identifies a point in the Process where a global Choreography or a Global Choreography Task is used. The Call Choreography acts as a place holder for the inclusion of the Choreography element it is calling. This pre-defined called Choreography element becomes a part of the definition of the parent Choreography.

A Call Choreography object shares the same shape as the Choreography Task and Sub-Choreography, which is a rectangle that has rounded corners, two or more Participant Bands, and an Activity Name Band. However, the target of what the Choreography Activity calls will determine the details of its shape.

- If the Call Choreography calls a Global Choreography Task, then the shape will be the same as a Choreography Task, but the boundary of the shape MUST have a thick line (see Figure 11.24).
- If the Call Choreography calls a Choreography, then there are two options:
  - The details of the called Choreography can be hidden and the shape will be the same as a collapsed Sub-Choreography, but the boundary of the shape MUST have a thick line (see Figure 11.25).
  - The details of the called Choreography can be shown and the shape will be the same as an expanded Sub-Choreography, but the boundary of the shape MUST have a thick line (see Figure 11.26).

![Figure 11.24 - A Call Choreography calling a Global Choreography Task](image)

![Figure 11.25 - A Call Choreography calling a Choreography (Collapsed)](image)
Figure 11.26 - A Call Choreography calling a Choreography (expanded)

The **Call Choreography** element inherits the attributes and model associations of **ChoreographyActivity** (see Figure 11.27 and Table 11.1). Table 11.4 presents the additional model associations of the **Call Choreography** element.
11.4.4 Global Choreography Task

A GlobalChoreographyTask is a reusable, atomic Choreography Task definition that can be called from within any Choreography by a Call Choreography.

The GlobalChoreographyTask element inherits the attributes and model associations of Collaboration (see Table 9.1), through its relationship to Choreography. Table 11.5 presents the additional model associations of the GlobalChoreographyTask element.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description/Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>calledChoreographyRef:</td>
<td>The element to be called, which will be either a Choreography or a GlobalChoreographyTask.</td>
</tr>
<tr>
<td>participantAssociations:</td>
<td>Specifies how Participants in a nested Choreography or GlobalChoreographyTask match up with the Participants in the Choreography referenced by the Call Choreography.</td>
</tr>
</tbody>
</table>

11.4.5 Looping Activities

Both Sub-Choreographies can have standard loops and multi-instances. Examples of Choreography Activities with the appropriate markers can be seen in Figure 11.12 and Figure 11.22.

11.4.6 The Sequencing of Activities

There are constraints on how Choreography Activities can be sequenced (through Sequence Flows) in a Choreography. These constraints are due to the limited visibility of the Participants, which only know of the progress of the Choreography by the Messages that occur. When a Participant sends or receives a Message, then that Participant knows exactly how far the Choreography has progressed. This means that the ordering of Choreography Activities need to take into account when the Participants send or receive Messages so that they Participants are NOT REQUIRED to guess about when it is their turn to send a Message.
The basic rule of **Choreography Activity** sequencing is this:

- The **Initiator** of a **Choreography Activity** MUST have been involved (as **Initiator** or **Receiver**) in the previous **Choreography Activity**.

Of course, the first **Choreography Activity** in a **Choreography** does not have this constraint.

Figure 11.28 shows a sequence of two **Choreography Activities** that follow this constraint. “Participant B” is the **Initiator** of “Choreography Task 2” after being the **Receiver** in “Choreography Task 1.” While there is no requirement that “Participant B” sends the **Message** immediately, since there can be internal work that the **Participant** needs to do prior to the **Message**. But in this situation there is no ambiguity that “Participant B” will be the **Initiator** of the next **Choreography Task**. “Participant C” does not know exactly when the **Message** will arrive from “Participant B,” but “Participant C” knows that one will arrive and there are not any additional requirements on the **Participant** until the **Message** arrives.

![Figure 11.28 - A valid sequence of Choreography Activities](image)

Naturally, the sequence of **Choreography Activities** shown in Figure 11.28, above can be expanded into a **Collaboration** diagram to show how the sequence can be enforced. Figure 11.29 shows the corresponding **Collaboration**. The diagram shows how the **Activities** within the individual **Pools** fit with the design of the **Choreography**.
When determining a valid sequence of Choreography Tasks, it is important to consider the type of Choreography Tasks that are being used. A single Choreography Task can be used for one or two Messages. Most of the time there will be one or two Messages for a Choreography Task. Figure 11.30 shows a sequence of Choreography Tasks, the first one being a two-way interaction, where the initiator sends a Message and gets a response from the other Participant.

Figure 11.29 - The corresponding Collaboration for a valid Choreography sequence

Figure 11.30 - A valid sequence of Choreography Activities with a two-way Activity
Figure 11.31 shows the corresponding Collaboration and how the two Choreography Tasks are reflected in the Processes within the Pools. The Choreography Task that has two Messages is reflected by three Process Tasks. Usually in these cases, the initiating Participant will use a single Activity to handle both the sending and receiving of the Messages. A BPMN Service Task can be used for this purpose and these types of Tasks are often referred to as “request-response” Tasks for Choreography modelers.

![Figure 11.31 - The corresponding Collaboration for a valid Choreography sequence with a two-way Activity](image)

Figure 11.32 shows how a sequence of Choreography Activities can be designed that would be invalid in the sense that an Initiating Participant would not know when the appropriate time would be to send a Message. In this example, “Participant A” is scheduled to send a Message to “Participant C” after “Participant B” sends a Message to “Participant C.” However, “Participant A” will not know when the Message from “Participant B” has been sent. So, there is no way to enforce the sequence that is modeled in the Choreography.

![Figure 11.32 - An invalid sequence of Choreography Activities](image)
Figure 11.33 shows the **Collaboration** view of the above **Choreography** diagram. It becomes clear that “Participant A” will not know the appropriate time to send **Message** “M3” to “Participant C.” If the **Message** is sent too soon, then “Participant C” will not be prepared to receive it. Thus, as a **Choreography**, the model in Figure 11.32, above, cannot be enforced.

---

**Figure 11.33 - The corresponding Collaboration for an invalid Choreography sequence**

### 11.5 Events

#### 11.5.1 Start Events

**Start Events** provide the graphical marker for the start of a **Choreography**. They are used much in the same way as they are used for a **Process** (see “Start Event” on page 238). This table shows how the types of **Start Events** are applied to **Choreography**.
### Table 11.6 – Use of Start Events in Choreography

<table>
<thead>
<tr>
<th>Type of Event</th>
<th>Usage in Choreography?</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Yes. This is really just a graphical marker since the arrival of the first <strong>Message</strong> in the <strong>Choreography</strong> is really the <strong>Trigger</strong> for the <strong>Choreography</strong>. <strong>Sub-Processes</strong>, however, we should look at. The <strong>Parent Process</strong> can be considered the <strong>trigger</strong>.</td>
</tr>
<tr>
<td>Message</td>
<td>No. A <strong>Message Start Event</strong>, in a stand-alone <strong>Choreography</strong>, has no way to show who the senders or receivers of the <strong>Message</strong> should be. A <strong>Choreography Task</strong> should be used instead. Thus, a <strong>None Start Event</strong> should be used as a graphical marker for the “start” of the <strong>Choreography</strong>.</td>
</tr>
<tr>
<td>Timer</td>
<td>Yes. All <strong>Participants</strong> have to have an agreement to the approximate time.</td>
</tr>
<tr>
<td>Escalation</td>
<td>No. An Escalation is only visible to a single <strong>Participant</strong>. That <strong>Participant</strong> will have to send a <strong>Message</strong> to the other <strong>Participants</strong>.</td>
</tr>
<tr>
<td>Error</td>
<td>No. An Error is only visible to a single <strong>Participant</strong>. That <strong>Participant</strong> will have to send a <strong>Message</strong> to the other <strong>Participants</strong>.</td>
</tr>
<tr>
<td>Compensation</td>
<td>No. Compensation is handled within a single <strong>Participant</strong> (an orchestration <strong>Process</strong>).</td>
</tr>
<tr>
<td>Conditional</td>
<td>Yes. This is actually determined internal to <strong>Participant</strong>, but then the other <strong>Participants</strong> know this has happened based the first interaction that follows.</td>
</tr>
<tr>
<td>Signal</td>
<td>Yes. The source of the <strong>Signal</strong> is NOT REQUIRED (and might not even be a <strong>Participant</strong> in the <strong>Choreography</strong>). There are no specific recipients of a <strong>Signal</strong>. All <strong>Participants</strong> of the <strong>Choreography</strong> (to comply) MUST be able to see the <strong>Signal</strong>.</td>
</tr>
<tr>
<td>Multiple</td>
<td>Yes. But they can only be Multiple <strong>Signals or Timers</strong>. As in <strong>Orchestration</strong>, this acts like an OR. Any one of the incoming <strong>Signals</strong> will Trigger the <strong>Choreography</strong>. Any <strong>Signal</strong> that follows would create a separate instance of the <strong>Choreography</strong>.</td>
</tr>
</tbody>
</table>
### 11.5.2 Intermediate Events

Table 11.7 - Use of Intermediate Events in Choreography

<table>
<thead>
<tr>
<th>Type of Event</th>
<th>Usage in Choreography?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>None</strong>: in Normal Flow</td>
<td>Yes. However, this really doesn’t have much meaning other than just documenting that a specific point has been reached in the <strong>Choreography</strong>. There would be no <strong>Message</strong> exchange or any delay in the <strong>Choreography</strong>.</td>
</tr>
<tr>
<td><strong>None</strong>: Attached to Activity boundary</td>
<td>No. There would be no way for <strong>Participants</strong> to know when the <strong>Activity</strong> should be interrupted.</td>
</tr>
<tr>
<td><strong>Message</strong>: in Normal Flow</td>
<td>No. A <strong>Message Intermediate Event</strong>, in a stand-alone <strong>Choreography</strong>, has no way to show who the senders or receivers of the <strong>Message</strong> should be. A <strong>Choreography Task</strong> should be used instead. Also, would the <strong>Event</strong> be a <strong>Catch</strong> or a <strong>Throw</strong>?</td>
</tr>
<tr>
<td><strong>Message</strong>: Attached to Activity boundary</td>
<td>Yes. Only for <strong>Choreography Tasks</strong>. The Intermediate Event has to be attached to the <strong>Participant Band</strong> of the receiver of the <strong>Message</strong> (since it is a catch Event). The sender of the message has to be the other <strong>Participant</strong> of the <strong>Choreography Task</strong>.</td>
</tr>
<tr>
<td><strong>Message</strong>: Use in Event Gateway</td>
<td>No. A <strong>Message Intermediate Event</strong>, in a stand-alone <strong>Choreography</strong>, has no way to show who the senders or receivers of the <strong>Message</strong> should be. A <strong>Choreography Task</strong> should be used instead.</td>
</tr>
<tr>
<td><strong>Timer</strong>: in Normal Flow</td>
<td>Yes. Time is not precise in <strong>Choreography</strong>. It is established by the last visible <strong>Choreography Activity</strong>. The <strong>Participants</strong> involved in the Choreography Activity that immediately precedes will have a rough approximation of the time—there will be no exact synchronization. For relative timers: Only the <strong>Participants</strong> involved in the Choreography Activity that immediately precedes the Event would know the time. The sender of the Choreography Activity that immediately follows the timer MUST be involved in the Choreography Activity that immediately precedes the timer. For absolute timers (full time/date): All <strong>Participants</strong> would know the time. There does not have to be a relationship between the <strong>Participants</strong> of the Choreography Activities that are on either side the timer. The sender of the Choreography Activity that immediately follows the timer is the <strong>Participant</strong> that enforces the timer.</td>
</tr>
</tbody>
</table>
### Table 11.7 - Use of Intermediate Events in Choreography

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Normal Flow</th>
<th>Activity Boundary</th>
<th>Event Gateway</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Timer</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Time is not exact in this case. It is established by the last visible Event. All Participants will have a rough approximation of the time—there will be no exact synchronization. This includes both interrupting and escalation Events. The Participants of the Choreography Activity that has the attached timer all enforce the timer. For relative timers: They all have to be involved in the Choreography Activity that immediately precedes the Activity with the attached timer. For absolute timers (full time/date): All Participants would know the time. They all have to be involved in the Choreography Activity that immediately precedes the Activity with the attached timer.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Timer</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>See Event-Based Gateway below.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Error</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>An Error is only visible to a single Participant. That Participant will have to send a Message to the other Participants.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Escalation</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>An Escalation is only visible to a single Participant. That Participant will have to send a Message to the other Participants.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Escalation</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>An Escalation is only visible to a single Participant. That Participant will have to send a Message to the other Participants.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cancel</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>These are Throw Events. As with a Message Event, there would be no indicator as to who is the source of the Cancel.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cancel</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>These are Catch Events. As with a Message Event, they would be attached to the Choreography Activity on the Participant Band that is receiving the Cancel. These would only be interrupting Events.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Compensation</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>These are Throw Events. As with a Message, there would be no indicator as to who is the source of the Cancel.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Compensation</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>These are Catch Events. As with a Message Event, they would be attached to the Choreography Activity on the Participant Band that is receiving the Cancel.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Conditional</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>This is a delay that waits for a change in data to trigger the Event. The data are to be visible to the Participants as it was data of a previously sent Message.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Conditional</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>This is an interruption that waits for a change in data to trigger the Event. The data are to be visible to the Participants as it was data of a previously sent Message.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Conditional</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>This is a delay that waits for a change in data to trigger the Event. The data are to be visible to the Participants as it was data of a previously sent Message.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
11.5.3 End Events

End Events provide a graphical marker for the end of a path within the Choreography.

Table 11.8 – Use of End Events in Choreography

<table>
<thead>
<tr>
<th>Type of Event</th>
<th>Usage in Choreography?</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Yes. This is really just a graphical marker since the sending of the previous Message in the Choreography is really the end of the Choreography. The Participants of the Choreography would understand that they would not expect any further Message at that point.</td>
</tr>
</tbody>
</table>
11.6 Gateways

In an Orchestration Process, Gateways are used to create alternative and/or parallel paths for that Process. Choreography has the same requirement of alternative and parallel paths. That is, interactions between Participants can happen in sequence, in parallel, or through exclusive selection. While the paths of Choreography follow the same basic patterns as that of an Orchestration Process, the lack of a central mechanism to maintain data visibility, and that there is no central evaluation, there are constraints as to how the Gateways are used in conjunction with the Choreography Activities that precede and follow the Gateways. These constraints are an extension of the basic sequencing constraints that was defined on page 335. The six (6) sections that follow will define how the types of Gateways are used in Choreography.
11.6.1 Exclusive Gateway

Exclusive Gateways (Decisions) are used to create alternative paths within a Process or a Choreography. For details of how Exclusive Gateways are used within an Orchestration Process see page 290.

Exclusive Gateways are used in Choreography, but they are constrained by the lack of a central mechanism to store the data that will be used in the Condition expressions of the Gateway’s outgoing Sequence Flows. Choreographies MAY contain natural language descriptions of the Gateway’s Conditions to document the alternative paths of the Choreography (e.g., “large orders” will go down one path while “small orders” will go down another path), but such Choreographies would be underspecified and would not be enforceable. To create an enforceable Choreography, the Gateway Conditions MUST be formal Condition Expressions. However:

- The data used for Gateway Conditions MUST have been in a Message that was sent prior to (upstream from) the Gateway.
- More specifically, all Participants that are directly affected by the Gateway MUST have either sent or received the Message(s) that contained the data used in the Conditions.
- Furthermore, all these Participants MUST have the same understanding of the data. That is, the actual values of the data cannot selectively change after a Participant has seen a Message. Changes to data during the course of the Choreography MUST be visible to all the Participants affected by the Gateway.

These constraints ensure that the Participants in the Choreography understand the basis (the actual value of the data) for the decision behind the Gateway.

One (1) or more Participants will actually “control” the Gateway decision; that is, these Participants make the decision through the internal Orchestration Processes. The decision is manifested by the particular Message that occurs in the Choreography (after the Gateway). This Message is the initiating Message of a Choreography Activity that follows theGateway. Thus, only the Participants that are the initiators of the Messages that follow the Gateway are the ones that control the decision. This means that:

- The initiating Participants of the Choreography Activities that follow the Gateway MUST have sent or received the Message that provided the data upon which the decision is made.
- The Message that provides the data for the Gateway MAY be in any Choreography Activity prior to the Gateway (i.e., it does not have to immediately precede Gateway).
Figure 11.34 - An example of the Exclusive Gateway

Figure 11.35 shows the Collaboration that demonstrates how the above Choreography that includes an Exclusive Gateway can be enforced.
Usually, the *initiators* for the *Choreography Activities* that follow the *Gateway* will be the same *Participant*. That is, there is only one *Participant* controlling the decision. Often, the receivers of the *initiating Message* for those Choreography Activities will be the same Participant. However, it is possible that there could be different Participants receiving the initiating Message for each Choreography Activity (see Figure 11.36).
Figure 11.36 - Different Receiving Choreography Activity Participants on the output sides of the Exclusive Gateway

This configuration can only be valid if all the Participants in the Choreography Activities that follow the Gateway have seen the data upon which the decision is made. If either “Participant B” or “Participant C” had not sent or received a Message with the appropriate data, then that Participant would not be able to know if they are suppose to receive a Message at that point in the Choreography. There is also the assumption that the value of the data is consistent from the point of view of all Participants.

Figure 11.37 displays the corresponding Collaboration view of the above Choreography Exclusive Gateway configuration.
Figure 11.37 - The corresponding Collaboration view of the above Choreography Exclusive Gateway configuration

The REQUIRED execution behavior of the **Gateway** and associated **Choreography Activities** are enforced through the **Business Processes** of the **Participants** as follows:

- Each **Choreography Activity** and the **Sequence Flow** connections are reflected in each **Participant Process**.
- The **Gateway** is reflected in the **Process** of each **Participant Process** that is an initiator of **Choreography Activities** that follow the **Gateway**.
- For the receivers in **Choreography Activities** that follow the **Gateway**, an **Event-Based Gateway** is used to consume the associated **Message** (sent as an outcome of the **Gateway**). When a **Participant** is the receiver of more than one of the alternative **Messages**, the corresponding receives follow the **Event-Based Gateway**. If the **Participant** is the receiver of only one such **Message**, that is also consumed through a receive following the **Event-Based Gateway**. This is because the **Participant Process** does not know whether it will receive a **Message** (since the **Gateway** entails a choice of outcomes).
11.6.2 Event-Based Gateway

As described above, the Event-Based Gateway represents a branching point in the Process where the alternatives are based on Events that occur at that point in the Process, rather than the evaluation of expressions using Process data. For details of how Event-Based Gateways are used within an Orchestration Process see “Event-Based Gateway” on page 297.

These Gateways are used in Choreography when the data used to make the decision is only visible to the internal Processes of one Participant. That is, there has been no Message sent within the Choreography that would expose the data used to make the decision. Thus, the only way that the other Participants can be aware of the results of the decision is by the particular Message that arrives next.

- On the right side of the Gateway: either
  - the senders MUST to be the same; or
  - the receivers MUST to be the same.
  - After the first Choreography Activity occurs, the other Choreography Activities for the Gateway MUST NOT occur.

- Message Intermediate Events MUST NOT be used in the Event-Based Gateway.
- Timer Intermediate Events MAY be used, but they restrict the participation in the Gateway.
  - For relative timers: All Participants on the right side of the Gateway MUST be involved in the Choreography Activity that immediately precedes the Gateway.
  - For absolute timers (full time/date): All Participants on the right side of the Gateway MUST be involved in the Choreography Activity that immediately precedes the Gateway.

- Signal Intermediate Events MAY be used (they are visible to all Participants).
- No other types of Intermediate Events are allowed.

Figure 11.38 - An example of an Event Gateway

Figure 11.39 displays the corresponding Collaboration view of the above Choreography Event Gateway configuration.
The REQUIRED execution behavior of the Event-Based Gateway and associated Choreography Activities are enforced through the Business Processes of the Participants as follows:

- Each Choreography Activity and the Sequence Flow connections is reflected in each Participant Process.
- If the senders following the Gateway are the same, the Event-Based Gateway is reflected as an Exclusive Gateway in that Participant’s Process. This is because the choice of which Message to send is determined by the same Participant. If the senders are different, sending occurs through different Processes.
- If the receivers are the same, the senders can be the same or different. In this case, the Event-Based Gateway is reflected in the receiver’s Process, with the different Message receives following the Gateway.
- If the receivers are different, the senders need to be the same. The Event-Based Gateway is reflected for different receiver Processes such that the respective receive follows the Gateway. A time-out can be used to ensure that the Gateway does not wait indefinitely.
11.6.3 Inclusive Gateway

Inclusive Gateways are used for modeling points of synchronization of a number of branches, not all of which are active, after which one or more alternative branches are chosen within a Choreography flow. For example, one of more branches MAY be activated upstream, in parallel, depending on the nature of goods in an order (e.g., large orders, fragile goods orders, orders belonging to pre-existing shipment contracts), and these are subsequently merged. The point of merge results in one or more risk mitigating outcomes (e.g., special insurance protection needed, special packaging needed, and different container categories needed). Inclusive Gateways are also used within an Orchestration Process see page 292.

Like Exclusive Gateways, Inclusive Gateways are used in a Choreography, but they are constrained by the lack of a central mechanism to store the data that will be used in the Condition expressions of the Gateway’s outgoing Sequence Flows. Choreographies MAY contain natural language descriptions of the Gateway’s Conditions to document the one more alternative paths of the Choreography (e.g., “special insurance protection needed,” “special packaging needed,” and different “container category needed”), but such Choreographies would be underspecified and would not be enforceable. To create an enforceable Choreography, the Gateway Conditions MUST be formal Condition Expressions. In general the following rules apply for the Expressions:

Like the enforceability of the Exclusive Gateway, the Inclusive Gateway in a Choreography requires that the data in the Expressions of the outgoing Sequence Flows of the Gateway be available to the initiators of the Choreography Activities of outgoing Sequence Flows. This means that the initiators of these Choreography Activities should also be senders or receivers of Messages in Choreography Activities immediately preceding the Gateway. The major difference, however, is that the synchronizing behavior of the Inclusive Gateway can only be enforced through one participant. Hence, the rules for enforceability are as follows:

- The data used for Gateway Conditions MUST have been in a Message that was sent prior to (upstream from) the Gateway.
- More specifically, all Participants that are directly affected by the Gateway MUST have either sent or received the Message(s) that contained the data used in the Conditions.
- Furthermore, all these Participants MUST have the same understanding of the data. That is, the actual values of the data cannot selectively change after a Participant has seen a Message. Changes to data during the course of the Choreography MUST be visible to all the Participants affected by the Gateway.
- Merge: In order to enforce the synchronizing merge of the Gateway, the sender of the Choreography Activity after the Gateway MUST participate in the Gateway immediately preceding the Gateway. This ensures that the merge can be enforced. (This relies on the assumption of logical atomicity of a Choreography Activity, otherwise the rule would require that all receivers are the same so that the Gateway is enforced in the receiver’s Process only).
- Split: In order to enforce the split side of the Gateway, the initiators of all Choreography Activities immediately following the Gateway MUST be the same as the common sender or receiver of Choreography Activities preceding the Gateway. The sender(s) of all the Choreography Activities after the Gateway MUST be involved in all the Choreography Activities that immediately precede the Gateway.

Figure 11.40 shows an example of a Choreography with an Inclusive Gateway. The Gateway is enforced in the corresponding Business Processes of the Participants involved. For the merge behavior to be enforced, the initiator of Choreography Activities immediately following the Gateway participates in the Choreography Activities immediately preceding the Gateway.
Figure 11.40 - An example of a Choreography Inclusive Gateway configuration
Figure 11.41 - The corresponding Collaboration view of the above Choreography Inclusive Gateway configuration

Figure 11.42, a variation of Figure 11.40 above, shows an example of a Choreography illustrating the enforcement of the split behavior of the Inclusive Gateway. For the split behavior to be enforced, the initiators of Choreography Activities immediately following the Gateway and the receiver of Choreography Activities immediately preceding the Gateway are the same Participant (i.e., A).
Figure 11.42 - An example of a Choreography Inclusive Gateway configuration
Figure 11.43 - The corresponding Collaboration view of the above Choreography Inclusive Gateway configuration
Figure 11.44 - Another example of a Choreography Inclusive Gateway configuration
Figure 11.45 - The corresponding Collaboration view of the above Choreography Inclusive Gateway configuration
11.6.4 Parallel Gateway

Parallel Gateways are used to create paths and are performed at the same time, within a Choreography flow. For details of how Parallel Gateways are used within an Orchestration Process see page 293.

Since there is no conditionality for these Gateways, they are available as-is in Choreography. They create parallel paths of the Choreography that all Participants are aware of.

- The sender(s) of all the Activities after the Gateway MUST be involved in all the Activities that immediately precede the Gateway.
- If there is a chain of Gateways with no Choreography Activities in between, the Choreography Activity that precedes the chain satisfies the above constraint.

Figure 11.46 shows the relationship of Choreography Activity Participants across the sides of the Parallel Gateway.

Figure 11.46 - The relationship of Choreography Activity Participants across the sides of the Parallel Gateway

Figure 11.47 shows the corresponding Collaboration view of the above Choreography Parallel Gateway configuration.
The REQUIRED execution behavior of the Parallel Gateway and associated Choreography Activities are enforced through the Business Processes of the Participants as follows:

- Each Choreography Activity and the Sequence Flow connections is reflected in each Participant Process.
- If the senders following the Parallel Gateway are the same, a Parallel Gateway is reflected in the sender’s Process followed by Message sending actions to the corresponding receivers.
- If the senders are different, the Parallel Gateway is manifested by Sequence Flows followed by the sending action in each Process.
11.6.5 Complex Gateway

Complex Gateways can model partial merges in Business Processes where when some but not all of a set of preceding branches complete, the Gateway fires. This can be considered the discriminator/n-of-m join pattern\(^1\) and is not supported through the inclusive OR merge since it is not concerned with sets of branches, but rather branches that have tokens. Applied in Choreographies, Complex Gateways can model tendering and information canvassing use cases where requests are sent to participants who respond at different times.

Consider an e-tender that sends a request for quote to multiple service providers (e.g., warehouse storage) in a marketplace. The e-tender Process sends out requests to each service provider and anticipates their response through three Choreography Activities. The response branches merge at a Complex Gateway to model the requirement that when 66% responses have arrived, an assessment of the tender can proceed. The assessment occurs after the Complex Gateway. If the assessment reports that the reserve amount indicated by the customer cannot be met, a new iteration of the tender is made. A key issue is to ensure that the responses should not be mixed across tender iterations. A Terminate End Event ensures that all Activities are terminated, when a tender has been successful.

![Figure 11.48 - An example of a Choreography Complex Gateway configuration](image-url)

---

11.6.6 Chaining Gateways

It is possible to chain Gateways. This means that a modeler can sequence two or more Gateways without any intervening Choreography Activities, however the constraints on what participants can appear before and after the chain MUST be observed.

11.7 Choreography within Collaboration

11.7.1 Participants

Participants are used in both Collaborations and Choreographies.
11.7.2 Swimlanes

Swimlanes, both Pools and Lanes, are not used in Choreographies. Pools are used exclusively in Collaborations (see page 114). Participants, which can be associated to Pools, however, are used in the Participant Bands of Choreography Tasks (see page 323) and Sub-Choreographies (see page 328). Pools can be used with Choreography diagrams when in the context of a Collaboration diagram (see page 362).

Lanes are not used in Choreography diagrams since Lanes are sub-partitions of a Pool and Choreographies are placed in between the Pools (if used in a Collaboration).

Figure 11.50 shows an example of a Choreography Process combined with Black Box Pools.

![Diagram showing a Choreography Process with Black Box Pools]

**Figure 11.50 - An example of a Choreography Process combined with Black Box Pools**

Figure 11.51 shows an example of a Choreography Process combined with Pools that contain Processes.
Figure 11.51 - An example of a Choreography Process combined with Pools that contain Processes

Choreography Task in Combined View

Sub-Choreography in Combined View

11.8 XML Schema for Choreography

Table 11.9 – Choreography XML schema

```xml
<xsd:element name="choreography" type="tChoreography" substitutionGroup="collaboration"/>
<xsd:complexType name="tChoreography">
  <xsd:complexContent>
    <xsd:extension base="tCollaboration">
      <xsd:sequence>
        <xsd:element ref="flowElement" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
Table 11.10 – GlobalChoreographyTask XML schema

```xml
<xsd:element name="globalChoreographyTask" type="tGlobalChoreographyTask"
    substitutionGroup="choreography"/>
<xsd:complexType name="tGlobalChoreographyTask">
    <xsd:complexContent>
        <xsd:extension base="tChoreography">
            <xsd:attribute name="initiatingParticipantRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 11.11 – ChoreographyActivity XML schema

```xml
<xsd:element name="choreographyActivity" type="tChoreographyActivity"/>
<xsd:complexType name="tChoreographyActivity" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tFlowNode">
            <xsd:sequence>
                <xsd:element name="participantRef" type="xsd:QName" minOccurs="2" maxOccurs="unbounded"/>
                <xsd:element name="correlationKey" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="initiatingParticipantRef" type="xsd:QName" use="required"/>
            <xsd:attribute name="loopType" type="tChoreographyLoopType" default="None"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 11.12 – ChoreographyTask XML schema

```xml
<xsd:element name="choreographyTask" type="tChoreographyTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tChoreographyTask">
    <xsd:complexContent>
        <xsd:extension base="tChoreographyActivity">
            <xsd:sequence>
                <xsd:element name="messageFlowRef" type="xsd:QName" minOccurs="1" maxOccurs="2"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```
Table 11.13 – CallChoreography XML schema

```xml
<xsd:element name="callChoreography" type="tCallChoreography" substitutionGroup="flowElement"/>
<xsd:complexType name="tCallChoreography">
  <xsd:complexContent>
    <xsd:extension base="tChoreographyActivity">
      <xsd:sequence>
        <xsd:element ref="participantAssociation" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="calledChoreographyRef" type="xsd:QName" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 11.14 – SubChoreography XML schema

```xml
<xsd:element name="subChoreography" type="tSubChoreography" substitutionGroup="flowElement"/>
<xsd:complexType name="tSubChoreography">
  <xsd:complexContent>
    <xsd:extension base="tChoreographyActivity">
      <xsd:sequence>
        <xsd:element ref="flowElement" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="artifact" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
12 BPMN Notation and Diagrams

12.1 BPMN Diagram Interchange (BPMN DI)

12.1.1 Scope

This chapter specifies the meta-model and schema for BPMN 2.0 Diagram Interchange (BPMN DI). The BPMN DI is meant to facilitate interchange of BPMN diagrams between tools rather than being used for internal diagram representation by the tools. The simplest interchange approach to ensure the unambiguous rendering of a BPMN diagram was chosen for BPMN DI. As such, BPMN DI does not aim to preserve or interchange any “tool smarts” between the source and target tools (e.g., layout smarts, efficient styling, etc.).

BPMN DI does not address or define the interchange of color information. The use of alternative colors in BPMN is non normative. The meaning or semantic of colors might vary from tool to tool or, from user to user, potentially leading to miss-interpretations.

BPMN DI does not ascertain that the BPMN diagram is syntactically or semantically correct.

12.1.2 Diagram Definition and Interchange

The BPMN DI meta-model, similar to the BPMN abstract syntax meta-model, is defined as a MOF-based meta-model. As such, its instances can be serialized and interchanged using XMI. BPMN DI is also defined by an XML schema. Thus its instances can also be serialized and interchanged using XML.

Both, BPMN DI meta-model and schema are harmonized with a draft version of the OMG Diagram Definition (DD) standard. Annex B contains the relevant parts of the referenced DD specifications that were used as foundation for the BPMN DI model and schema. The provided DD contains two main parts: the Diagram Commons (DC) and the Diagram Interchange (DI). The DC defines common types like bounds and fonts, while the DI provides a framework for defining domain specific diagram models. As a domain specific DI, BPMN DI defines a few new meta-model classes that derive from the abstract classes from DI.

The focus of BPMN DI is the interchange of laid out shapes and edges that constitute a BPMN diagram. Each shape and edge references a particular BPMN model element. The referenced BPMN model elements are all part of the actual BPMN model. As such, BPMN DI is meant to only contain information that is neither present, nor derivable, from the BPMN model whenever possible. Simply put, to render a BPMN diagram both the BPMN DI instance(s) and the referenced BPMN model are REQUIRED.

From the BPMN DI perspective, a BPMN diagram is a particular snapshot of a BPMN model at a certain point in time. Multiple BPMN diagrams can be exchanged referencing model elements from the same BPMN model. Each diagram may provide an incomplete or partial depiction of the content of the BPMN model. BPMN DI does not ascertain that the BPMN diagram is syntactically or semantically correct.

As described in Section 15, a BPMN model package consists of one or more files. Each file may contain any number of BPMN diagrams. The exporting tool is free to decide how many diagrams are exported and the importing tool is free to decide if and how to present the contained diagrams to the user.
12.1.3 How to Read this Chapter

The normative BPMN 2.0 Diagram Interchange (BPMN DI) specification has three parts. Section 12.2 defines BPMN DI; an instance of the DI meta-model provided at Annex B. Section 12.3 provides a library of the BPMN element depictions and an unambiguous resolution between a referenced BPMN model element and its depiction. Finally, Section 12.4 provides examples to support the interpretation of the specification. Some BPMN diagram depictions along with their XML BPMN DI serializations are provided.

12.2 BPMN Diagram Interchange (DI) Meta-model

12.2.1 Overview

The BPMN DI is an instance of the DI meta-model provided at Annex B. The basic concept of BPMN DI, as with DI in general, is that serializing a diagram [BPMNDiagram] for interchange requires the specification of a collection of shapes [BPMNShape] and edges [BPMNEdge] on a plane [BPMNPlane].

BPMNPlane, BPMNShape, and BPMNEdge MUST reference exactly one abstract syntax BPMN element from the BPMN model using the bpmnElement attribute. The only exception is for a Data Association connected to a Sequence Flow (See Figure 10.68). This is a visual short cut that actually normalizes two Data Associations within the BPMN model. In this case, the resolution is made from the BPMN DI attributes rather than the abstract syntax reference [bpmnElement] (See Table 12.35 - Depiction Resolution for Connecting Objects).

The BPMN DI classes only define the visual properties used for depiction. All other properties that are REQUIRED for the unambiguous depiction of the BPMN element are derived from the referenced bpmnElement.

Multiple depictions of a specific BPMN element in a single diagram is NOT allowed, except for Participants in a choreography (i.e., Participant Bands). For example, it is not allowed to depict a Task twice in the same diagram, but it is allowed to depict the same Task in two different diagrams.

BPMN diagrams may be an incomplete or partial depiction of the content of the BPMN model. Some BPMN elements from a BPMN model may not be present in any of the diagram instances being interchanged.

BPMN DI does not provide for any containment concept. The BPMNPlane is an ordered collection of mixed BPMNShape(s) and BPMNEdge(s). The order of the BPMNShape(s) and BPMNEdge(s) inside a BPMNPlane determines their Z-order (i.e., what is in front of what). BPMNShape(s) and BPMNEdge(s) that are meant to be depicted “on top” of other BPMNShape(s) and BPMNEdge(s) MUST appear after them in the BPMNPlane. Therefore, the exporting tool MUST order all BPMNShape(s) and BPMNEdge(s) such that the desired depiction can be rendered.

12.2.2 Abstract Syntax

This section introduces the Abstract Syntax of BPMN DI. BPMN DI is an instance of the DI meta-model provided at Annex B.
Figure 12.1 - BPMN Diagram

Figure 12.2 - BPMN Plane

Figure 12.3 - BPMN Shape
12.2.3 Classifier Descriptions

12.2.3.1 BPMNDiagram [Class]

BPMNDiagram is a kind of diagram that depicts all or part of a BPMN model.

Description

BPMNDiagram represents a depiction of all or part of a BPMN model. It specializes DI::Diagram and redefines the root element (the top most diagram element) to be of type BPMNPlane. A BPMN diagram can also own a collection of BPMNStyle elements that are referenced by BPMNLabel elements in the diagram. These style elements represent the unique appearance styles used in the diagram.

Abstract Syntax

- Figure 12.1 - BPMN Diagram
Generalizations

• DI::Diagram

Associations

• + plane : BPMNPlane [1] {redefines rootElement}
  a BPMN plane element that is the container of all diagram elements in this diagram.

• + labelStyle : BPMNLabelStyle [*] {subsets style}
  a collection of BPMN label styles that are owned by the diagram and referenced by label elements.

Table 12.1 – BPMNDiagram XML schema

<xsd:complexType name="BPMNDiagram">
  <xsd:complexContent>
    <xsd:extension base="di:Diagram">
      <xsd:sequence>
        <xsd:element ref="bpmndi:BPMNPlane"/>
        <xsd:element ref="bpmndi:BPMNLabelStyle" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

12.2.3.2 BPMNPlane [Class]

A BPMNPlane is the BPMNDiagram container of BPMNShape and BPMNEdge.

Description

A BPMNPlane specializes DI::Plane and redefines its model element reference to be of type (BPMN) BaseElement. A BPMNPlane can only reference a BaseElement of the types: Process, SubProcess, AdHocSubProcess, Transaction, Collaboration, Choreography or SubChoreography.

BPMNPlane element is always owned by a BPMNDiagram and represents the root diagram element of that diagram. The plane represents a 2 dimensional surface with an origin at (0, 0) along the x and y axes with increasing coordinates to the right and bottom. Only positive coordinates are allowed for diagram elements that are nested in a BPMNPlane. This means that the union of all the nested elements' bounds is deemed to be located at the plane's origin point.

Abstract Syntax

• Figure 12.1 - BPMN Diagram
• Figure 12.2 - BPMN Plane

Generalizations

• DI::Plane
Associations

- + bpmnElement : BaseElement [0..1] {redefines modelElement}  
  a reference to either a Process, SubProcess, AdHocSubProcess, Transaction, Collaboration, Choreography or SubChoreography in a BPMN model.

Table 12.2 – BPMNPlane XML schema

```xml
<xsd:complexType name="BPMNPlane">
  <xsd:complexContent>
    <xsd:extension base="di:Plane">
      <xsd:attribute name="bpmnElement" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

12.2.3.3 BPMNShape [Class]

BPMNShape is a kind of shape that can depict a BPMN model element.

Description

BPMNShape represents a depiction of a (typically a node) BPMN model element. It specializes DI::LabeledShape and redefines its model element reference to be of type (BPMN) BaseElement, allowing it to reference an element from a BPMN model.

BPMNShape also contains an optional label of type BPMNLabel that can be nested in the shape when it has a visible textual label with its own bounding box.

The shape also contains a number of normative notational options that can be specified for different types of BPMN elements depicted by the shape. Those options, each represented by a separate property, and described below, allow for recording the specific notational style desired for the shape.

All BPMNShape elements are owned directly by a BPMNPlane (that is the root element in a BPMNDiagram), i.e., shapes are not nested within each other in the BPMN DI model although they may appear that way when depicted. The bounds of a BPMNShape are always relative to that plane’s origin point and are REQUIRED to be positive coordinates. Note that the bounds’ x and y coordinates are the position of the upper left corner of the shape (relative to the upper left corner of the plane).

Abstract Syntax

- Figure 12.3 - BPMN Shape
- Figure 12.4 - BPMN Edge

Generalizations

- DI::LabeledShape
Attributes

- + isHorizontal : Boolean [0..1]
  
an optional attribute that should be used only for Pools and Lanes. It determines if it should be depicted horizontally (true) or vertically (false).

- + isExpanded : Boolean [0..1]
  
an optional attribute that should be used only for SubProcess, AdHocSubProcess, Transaction, SubChoreographies, CallActivities and CallChoreographies. It determines if it should be depicted expanded (true) or collapsed (false).

- + isMarkerVisible : Boolean [0..1]
  
an optional attribute that should be used only for Exclusive Gateway. It determines if the marker should be depicted on the shape (true) or not (false).

- + participantBandKind : ParticipantBandKind [0..1]
  
an optional attribute that should only be used for Participant Bands. If this attribute is present, it means that the participant should be depicted as a Participant Band instead of as a Pool.

- + isMessageVisible : Boolean [0..1]
  
an optional attribute that should only be used for Participant Bands. It determines if an envelope decorator should be depicted linked to the Participant Band.

- + choreographyActivityShape : BPMNShape [0..1]
  
an optional attribute that should only be used for Participant Bands. It is REQUIRED for a BPMNShape depicting a Participant Band. This is REQUIRED in order to relate the Participant Band to the BPMNShape depicting the Choreography Activity that this Participant Band is related to.

Associations

- + bpmnElement : BaseElement [0..1] {redefines modelElement}
  
a reference to a BPMN node element that this shape depicts. Note that although optional a bpmnElement must be provided for a BPMNShape.

- + label : BPMNLabel [0..1] {subsets ownedLabel}
  
an optional label that is nested when the shape has a visible text label with its own bounding box.
12.2.3.4 ParticipantBandKind [Enumeration]

ParticipantBandKind defines the type of Participant Band to depict.

**Description**

Participant bands can be depicted in 3 ways:

1. a top band is rectangular with rounded corners at the top
2. a middle band is rectangular
3. a bottom band is rectangular with rounded corners at the bottom

Participant bands can be depicted in 2 shadings:

1. initiating (the band should not be shaded)
2. non_initiating (the band should be shaded)

**Abstract Syntax**

- Figure 12.3 - BPMN Shape

**Literals**

- top_initiating - the band should be depicted as a non shaded top band
- middle_initiating - the band should be depicted as a non shaded middle band
- bottom_initiating - the band should be depicted as a non shaded bottom band
- top_non_initiating - the band should be depicted as a shaded top band
- middle_non_initiating - the band should be depicted as a shaded middle band
- bottom_non_initiating - the band should be depicted as a shaded bottom band
12.2.3.5 BPMNEdge [Class]

BPMNEdge is a kind of edge that can depict a relationship between two BPMN model elements.

Description

BPMNEdge represents a depiction of a relationship between two (source and target) BPMN model elements. It specializes DI::LabeledEdge and redefines its model element reference to be of type (BPMN) BaseElement, allowing it to reference a relationship element from a BPMN model.

BPMNEdge also redefines its source and target references to be of type DiagramElement (either BPMNShape or BPMNEdge).

The source or target definition should only be present if the edge is depicted between a different source or target than the one referenced by the BPMN model element of the BPMNEdge. Only the different source or target is REQUIRED. Both attributes should be present only if both are different. This is the case, for instance, if a Message Flow target is not depicted in the current diagram because it is inside a black box Pool. The Message Flow could then define its target as being the BPMNShape depicting the Pool to connect it to the boundary of that black box Pool.

BPMNEdge also contains an optional label of type BPMNLabel that can be nested in the edge when it has a visible textual label with its own bounding box.

All BPMNEdge elements are owned directly by a BPMNPlane (that is the root element in a BPMNDiagram). The waypoints of BPMNEdge are always relative to that plane's origin point and are REQUIRED to be positive coordinates.

Abstract Syntax

- Figure 12.4 - BPMN Edge

Generalizations

- DI::LabeledEdge

Associations

- + label : BPMNLabel [0..1] {subsets ownedLabel}
  an optional label that is nested when the edge has a visible text label with its own bounding box.

- + bpmnElement : BaseElement [0..1] {redefines modelElement}
  a reference to a connecting BPMN element that this edge depicts. Note that this reference is only optional for the specific case of a Data Association connected to a Sequence Flow; in all other cases a referenced element must be provided.

- + sourceElement : DiagramElement [0..1] {redefines source}
  an optional reference to the edge’s source element if it is different from the source inferred from the bpmnElement association.

- + targetElement : DiagramElement [0..1] {redefines target}
  an optional reference to the edge’s target element if it is different from the target inferred from the bpmnElement association.

- messageVisibleKind : MessageVisibleKind [0..1]
  an optional attribute that should be used only for Message Flow. It determines if an envelope decorator should be depicted and the kind of envelope to be depicted.
Table 12.4 – BPMNEdge XML schema

```xml
<xsd:complexType name="BPMNEdge">
  <xsd:complexContent>
    <xsd:extension base="di:LabeledEdge">
      <xsd:sequence>
        <xsd:element ref="bpmndi:BPMNLabel" minOccurs="0" />
      </xsd:sequence>
      <xsd:attribute name="bpmnElement" type="xsd:QName" />
      <xsd:attribute name="sourceElement" type="xsd:QName" />
      <xsd:attribute name="targetElement" type="xsd:QName" />
      <xsd:attribute name="messageVisibleKind" type="bpmndi:MessageVisibleKind" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

12.2.3.6 MessageVisibleKind [Enumeration]

MessageVisibleKind defines the type of envelope that is visible.

Description
MessageVisibleKind is applicable only to Participant Band and Message Flow.

For Message Flow, the envelope should be positioned in the middle of the edge.

For Participant Band, the envelope should be positioned over (for top band) or under (for bottom band) and connected to the band using an association. Note that only Choreography Task Participant Bands are allowed to show the envelope. Middle bands being only used for a SubChoreography can thus not have envelope showing.

Abstract Syntax
- Figure 12.3 - BPMN Shape

Literals
- initiating - The envelope should not be shaded.
- non_initiating - The envelope should be shaded.

12.2.3.7 BPMNLabel [Class]

BPMNLabel is a kind of label that depicts textual info about a BPMN element.

Description
BPMNLabel represents a depiction of some textual information about a BPMN element. It specializes DI::Label and redefines its style reference to be of type BPMNLabelStyle, which contains information about the appearance of the label (e.g., the chosen font). The referenced style is owned by the diagram that nests the label.

A BPMN label is not a top-level element but is always nested inside either a BPMNShape or a BPMNEdge. It does not have its own reference to a BPMN element but rather inherits that reference (if any) from its parent shape or edge. The textual info depicted by the label is derived from that referenced BPMN element.
The bounds of BPMNLabel are always relative to the containing plane’s origin point. Note that the bounds’ x and y coordinates are the position of the upper left corner of the label (relative to the upper left corner of the plane).

Abstract Syntax

- Figure 12.3 - BPMN Shape
- Figure 12.4 - BPMN Edge
- Figure 12.5 - BPMN Label

Generalizations

- DI::Label

Associations

- + labelStyle : BPMNLabelStyle [0..1] {redefines style}  
  an optional reference to a label style (owned by the diagram) that gives the appearance options for the label. If not specified, the style of the label can be assumed by a tool.

Table 12.5 – BPMNLabel XML schema

```
<xsd:complexType name="BPMNLabel">
  <xsd:complexContent>
    <xsd:extension base="di:Label">
      <xsd:attribute name="labelStyle" type="xsd:QName" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

12.2.3.8 BPMNLabelStyle [Class]

BPMNLabelStyle is a kind of style that gives the appearance options for a BPMNLabel.

Description

BPMNLabelStyle represents the appearance options for elements of type BPMNLabel. It specializes DI::Style and contains a description of a font that is used in depicting a BPMNLabel. One or more labels may reference the same BPMNLabelStyle element, which must be owned by a BPMNDiagram.

Abstract Syntax

- Figure 12.1 - BPMN Diagram
- Figure 12.5 - BPMN Label

Generalizations

- DI::Style

Attributes

- + font : Font[1] - a font object that describes the properties of the font used for depicting the labels that reference this style.
12.2.4 Complete BPMN DI XML Schema

Table 12.6 – BPMNLabelStyle XML schema

```xml
<xsd:complexType name="BPMNLabelStyle">
  <xsd:complexContent>
    <xsd:extension base="di:Style">
      <xsd:sequence>
        <xsd:element ref="dc:Font"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 12.7 - Complete BPMN DI XML schema

```xml
  <xsd:element name="BPMNDiagram" type="bpmndi:BPMNDiagram" />
  <xsd:element name="BPMNPlane" type="bpmndi:BPMNPlane" />
  <xsd:element name="BPMNLabelStyle" type="bpmndi:BPMNLabelStyle" />
  <xsd:element name="BPMNShape" type="bpmndi:BPMNShape" substitutionGroup="di:DiagramElement" />
  <xsd:element name="BPMNLabel" type="bpmndi:BPMNLabel" />
  <xsd:element name="BPMNEdge" type="bpmndi:BPMNEdge" substitutionGroup="di:DiagramElement" />
  <xsd:complexType name="BPMNDiagram">
    <xsd:complexContent>
      <xsd:extension base="di:Diagram">
        <xsd:sequence>
          <xsd:element ref="bpmndi:BPMNPlane" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```
Table 12.7 – Complete BPMN DI XML schema (continued)

```xml
<xsd:complexType name="BPMNPlane">
  <xsd:complexContent>
    <xsd:extension base="di:Plane">
      <xsd:attribute name="bpmnElement" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="BPMNEdge">
  <xsd:complexContent>
    <xsd:extension base="di:LabeledEdge">
      <xsd:sequence>
        <xsd:element ref="bpmndi:BPMNLabel" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="bpmnElement" type="xsd:QName"/>
      <xsd:attribute name="sourceElement" type="xsd:QName"/>
      <xsd:attribute name="targetElement" type="xsd:QName"/>
      <xsd:attribute name="messageVisibleKind" type="bpmndi:MessageVisibleKind"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="BPMNShape">
  <xsd:complexContent>
    <xsd:extension base="di:LabeledShape">
      <xsd:sequence>
        <xsd:element ref="bpmndi:BPMNLabel" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="bpmnElement" type="xsd:QName"/>
      <xsd:attribute name="isHorizontal" type="xsd:boolean"/>
      <xsd:attribute name="isExpanded" type="xsd:boolean"/>
      <xsd:attribute name="isMarkerVisible" type="xsd:boolean"/>
      <xsd:attribute name="isMessageVisible" type="xsd:boolean"/>  
      <xsd:attribute name="participantBandKind" type="bpmndi:ParticipantBandKind"/>
      <xsd:attribute name="choreographyActivityShape" type="xsd:QName"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
12.3 Notational Depiction Library and Abstract Element Resolutions

As a notation, BPMN specifies the depiction for each of the BPMN elements.

Serializing a BPMN diagram for interchange requires the specification of a collection of BPMNShape(s) (see page 372) and BPMNEdge(s) (see page 375) on the BPMNPlane (see page 371) of the BPMNDiagram (see page 370). The BPMNShape(s) and BPMNEdge(s) attributes must be populated in such a way as to allow the unambiguous rendering of the BPMN diagram by the receiving party. More specifically, the BPMNShape(s) and BPMNEdge(s) must reference...
BPMN model element [bpmnElement]. If no bpmnElement is referenced or if the reference is invalid, it is expected that this shape or edge should not be depicted. The only exception is for a Data Association connected to a Sequence Flow (See Figure 10.68). This is a visual short cut that actually normalizes two Data Associations within the BPMN model. In this case, the resolution is made from the BPMN DI attributes rather than the abstract syntax reference [bpmnElement] (See Table 12.35 - Depiction Resolution for Connecting Objects).

When rendering a BPMN diagram, the correct depiction of a BPMNShape or BPMNEdge depends mainly on the referenced BPMN model element [bpmnElement] and its particular attributes and/or references.

The purpose of this section is to: provide a library of the BPMN element depictions, and to provide an unambiguous resolution between the referenced BPMN model element [bpmnElement], BPMNShape or BPMNEdge and their depiction. Depiction resolution tables are provided below for both BPMNShape (Section 12.3.2) and BPMNEdge (Section 12.3.3).

### 12.3.1 Labels

Both BPMNShape and BPMNEdge may have labels (e.g., its name) placed inside the shape/edge, or above or below the shape/edge, in any direction or location, depending on the preference of the modeler or modeling tool vendor.

Labels are optional for BPMNShape and BPMNEdge. When there is a label, the position of the label is specified by the bounds of the BPMNLabel of the BPMNShape or BPMNEdge. Simply put, label visibility is defined by the presence of the BPMNLabel element. The bounds of the BPMNLabel are optional and always relative to the containing BPMNPlane's origin point (see page 376). The depiction resolution tables provided below exemplify default label positions for BPMNShape kinds (Section 12.3.2) and BPMNEdge kinds (Section 12.3.3) if no BPMNLabel bounds are provided.

The text of the label to be rendered is obtained by resolving the name attribute of the referenced BPMN model element [bpmnElement] from the BPMNShape or BPMNEdge. In the particular case when the referenced BPMN model element [bpmnElement] is a DataObjectReference, the text of the label to be rendered is obtained by concatenating the name attribute of the referenced BPMN model element [bpmnElement] and the name attribute of the dataState attribute of this DataObjectReference (see Figure 12.6 - Depicting a Label for a DataObjectReference with its state).

![Label](image)

**Figure 12.6 - Depicting a Label for a DataObjectReference with its state**

The properties of the font to be used for rendering the label are optional and provided by the labelStyle of the BPMNLabel. If not provided, the tool should use its default style to depict the label.

### 12.3.2 BPMNShape

#### Markers for Activities

Various BPMN Activities can be decorated with markers at the bottom center of the shape.
Loop Characteristic markers may need to be rendered when the referenced BPMN model element [bpmnElement] of a BPMNShape is a Task, ServiceTask, SendTask, ReceiveTask, UserTask, ManualTask, BusinessRuleTask, ScriptTask, SubProcess, AdHocSubProcess, Transaction or CallActivity. Note that Loop Characteristic Markers (Loop, Multi-Instance - Parallel and Multi-Instance - Sequential) are mutually exclusive markers. That is, only one of them can be present on a single shape. See Table 10.8 - Depiction Resolution for Loop Characteristic Markers. Note that the patterns of Markers depicted in Table 10.8 also apply to Transaction and Call Activity which have different border depictions (i.e., double border or thick border).

A Compensation marker may need to be rendered when the referenced BPMN model element [bpmnElement] of a BPMNShape is a Task, ServiceTask, SendTask, ReceiveTask, UserTask, ManualTask, BusinessRuleTask, ScriptTask, SubProcess, AdHocSubProcess, Transaction or CallActivity. See Table 12.8 - Depiction Resolution for Compensation Marker.

In the case of expandable kind of shapes, the markers (Compensation or Loop Characteristic) are placed to the left of the + on the shape.

The Compensation marker may be combined with a Loop Characteristic Marker. All the markers that are present must be grouped and the whole group centered to the bottom of the shape. See Figure 12.7 - Combined Compensation and Loop Characteristic Marker Example.

Note that in the case where the referenced BPMN model element [bpmnElement] of a BPMNShape is an AdHocSubProcess, the shape has its tilde marker to the right of the + (See page 386).

### Table 12.8 - Depiction Resolution for Loop Compensation Marker

<table>
<thead>
<tr>
<th>Loop Characteristic Marker:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Loop</td>
<td><img src="Diagram" alt="Diagram" /></td>
<td>[bpmnElement: Task, ServiceTask, SendTask, ReceiveTask, UserTask, ManualTask, BusinessRuleTask, ScriptTask, SubProcess, AdHocSubProcess, Transaction or CallActivity] where loopCharacteristics is of type StandardLoopCharacteristics</td>
</tr>
</tbody>
</table>

**Table 12.8 - Depiction Resolution for Loop Compensation Marker**
Table 12.8 - Depiction Resolution for Loop Compensation Marker

<table>
<thead>
<tr>
<th>Loop Characteristic Marker:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multi-Instance - Parallel</strong></td>
<td><img src="image" alt="Diagram" /></td>
<td>[Task, ServiceTask, SendTask, ReceiveTask, UserTask, ManualTask, BusinessRuleTask, ScriptTask, SubProcess, AdHocSubProcess, Transaction or CallActivity] where loopCharacteristics is of type MultipleLoopCharacteristics with attribute isSequential to false.</td>
</tr>
<tr>
<td><strong>Multi-Instance - Sequential</strong></td>
<td><img src="image" alt="Diagram" /></td>
<td>[Task, ServiceTask, SendTask, ReceiveTask, UserTask, ManualTask, BusinessRuleTask, ScriptTask, SubProcess, AdHocSubProcess, Transaction or CallActivity] where loopCharacteristics is of type MultipleLoopCharacteristics with attribute isSequential to true.</td>
</tr>
<tr>
<td><strong>BPMNShape Attributes:</strong></td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
Tasks [BPMNShape]

There are different types of Tasks identified within BPMN. The specific Task type depiction is obtained by placing a Task type maker in the upper left corner of the Task shape. A Task that is no further specified is called an Abstract Task.

Tasks (Abstract, Service, Send, Receive, User, Manual, Business Rule or Script) can also have Compensation and/or Loop Characteristic markers at the bottom center of the shape as defined above (see page 381).
<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
<th>bpmnElement:</th>
<th>BPMNShape Attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Task</td>
<td><img src="image" alt="Label" /></td>
<td>Task</td>
<td>Task</td>
<td>None</td>
</tr>
<tr>
<td>Service Task</td>
<td><img src="image" alt="Label" /></td>
<td>ServiceTask</td>
<td>ServiceTask</td>
<td>None</td>
</tr>
<tr>
<td>Send Task</td>
<td><img src="image" alt="Label" /></td>
<td>SendTask</td>
<td>SendTask</td>
<td>None</td>
</tr>
<tr>
<td>Receive Task</td>
<td><img src="image" alt="Label" /></td>
<td>ReceiveTask</td>
<td>ReceiveTask</td>
<td>None</td>
</tr>
<tr>
<td>User Task</td>
<td><img src="image" alt="Label" /></td>
<td>UserTask</td>
<td>UserTask</td>
<td>None</td>
</tr>
<tr>
<td>Manual Task</td>
<td><img src="image" alt="Label" /></td>
<td>ManualTask</td>
<td>ManualTask</td>
<td>None</td>
</tr>
<tr>
<td>Business Rule Task</td>
<td><img src="image" alt="Label" /></td>
<td>BusinessRuleTask</td>
<td>BusinessRuleTask</td>
<td>None</td>
</tr>
<tr>
<td>Script Task</td>
<td><img src="image" alt="Label" /></td>
<td>ScriptTask</td>
<td>ScriptTask</td>
<td>None</td>
</tr>
</tbody>
</table>
Collapsed Sub-Processes [BPMNShape]

Collapsed Sub-Processes can also have Compensation and/or Loop Characteristic markers at the bottom center of the shape as defined above (see page 381).

| Table 12.10 – Depiction Resolution for Collapsed Sub-Processes |
|:-------------------|:-------------------|:-------------------|
| Kind:              | Depiction:         | Specific Depiction Resolution: |
| Sub-Process -       |                   | bpmnElement:          |
|        Collapsed    |                   | SubProcess where     |
|                   |                   | triggeredByEvent is false. |
|                   |                   | None or              |
|                   |                   | isExpanded is        |
|                   |                   | false               |

Expanded Sub-Processes [BPMNShape]

Expanded Sub-Processes can also have Compensation and/or Loop Characteristic markers at the bottom center of the shape as defined above (see page 381).

| Table 12.11 – Depiction Resolution for Expanded Sub-Processes |
|:-------------------|:-------------------|:-------------------|
| Kind:              | Depiction:         | Specific Depiction Resolution: |
| Sub-Process -       |                   | bpmnElement:          |
|        Expanded     |                   | SubProcess where     |
|                   |                   | triggeredByEvent is false. |
|                   |                   | isExpanded is        |
|                   |                   | true                |

Collapsed Ad Hoc Sub-Processes [BPMNShape]

Collapsed Ad Hoc Sub-Processes can also have a Compensation marker at the bottom center of the shape as defined above (see page 381).
Expanded Ad Hoc Sub-Processes [BPMNShape]

Expanded Ad Hoc Sub-Processes can also have a Compensation marker at the bottom center of the shape as defined above (see page 381).

Collapsed Transactions [BPMNShape]

Collapsed Transactions can also have Compensation and/or Loop Characteristic markers at the bottom center of the shape as defined above (see page 381).

---

**Table 12.12 – Depiction Resolution for Collapsed Ad Hoc Sub-Processes**

<table>
<thead>
<tr>
<th>Kind: Ad Hoc Sub-Process - Collapsed</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image" alt="Label" /></td>
<td>bpmnElement: AdHocSubProcess Attributes: None or isExpanded is false</td>
</tr>
</tbody>
</table>

**Table 12.13 – Depiction Resolution for Expanded Ad Hoc Sub-Processes**

<table>
<thead>
<tr>
<th>Kind: Ad Hoc Sub-Process - Expanded</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image" alt="Label" /></td>
<td>bpmnElement: AdHocSubProcess Attributes: None or isExpanded is true</td>
</tr>
</tbody>
</table>

**Table 12.14 – Depiction Resolution for Collapsed Transactions**

<table>
<thead>
<tr>
<th>Kind: Transaction - Collapsed</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image" alt="Label" /></td>
<td>bpmnElement: Transaction Attributes: None or isExpanded is false</td>
</tr>
</tbody>
</table>
Expanded Transactions [BPMNShape]

Expanded Transactions can also have Compensation and/or Loop Characteristic markers at the bottom center of the shape as defined above (see page 381).

Table 12.15 – Depiction Resolution for Tasks

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transaction - Expanded</strong></td>
<td><img src="image" alt="Label" /></td>
<td><strong>bpmnElement:</strong> Transaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>BPMNShape Attributes:</strong> None or isExpanded is true</td>
</tr>
</tbody>
</table>

Collapsed Event Sub-Processes [BPMNShape]

Table 12.16 – Depiction Resolution for Collapsed Event Sub-Processes

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Non-interrupting Message - Event Sub-Process - Collapsed</strong></td>
<td><img src="image" alt="Label" /></td>
<td><strong>bpmnElement:</strong> SubProcess where triggeredByEvent is true and the one-and-only start event has one EventDefinition of type MessageEventDefinition and isInterrupting is false.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>BPMNShape Attributes:</strong> None or isExpanded is false</td>
</tr>
<tr>
<td><strong>Interrupting - Message - Event Sub-Process - Collapsed</strong></td>
<td><img src="image" alt="Label" /></td>
<td><strong>bpmnElement:</strong> SubProcess where triggeredByEvent is true and the one-and-only start event has one EventDefinition of type MessageEventDefinition and isInterrupting is true.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>BPMNShape Attributes:</strong> None or isExpanded is false</td>
</tr>
<tr>
<td><strong>Non-interrupting - Timer - Event Sub-Process - Collapsed</strong></td>
<td><img src="image" alt="Label" /></td>
<td><strong>bpmnElement:</strong> SubProcess where triggeredByEvent is true and the one-and-only start event has one EventDefinition of type TimerEventDefinition and isInterrupting is false.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>BPMNShape Attributes:</strong> None or isExpanded is false</td>
</tr>
</tbody>
</table>
### Table 12.16 – Depiction Resolution for Collapsed Event Sub-Processes

<table>
<thead>
<tr>
<th>Sub-Process Description</th>
<th>Depiction</th>
<th>Description</th>
<th>None or isExpanded is false</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupting - Timer - Event Sub-Process - Collapsed</td>
<td><img src="image" alt="Label" /></td>
<td>SubProcess where triggeredByEvent is true and the one-and-only start event has one EventDefinition of type TimerEventDefinition and isInterrupting is true.</td>
<td>None or isExpanded is false</td>
</tr>
<tr>
<td>Non-interrupting - Conditional - Event Sub-Process - Collapsed</td>
<td><img src="image" alt="Label" /></td>
<td>SubProcess where triggeredByEvent is true and the one-and-only start event has one EventDefinition of type ConditionalEventDefinition and isInterrupting is false.</td>
<td>None or isExpanded is false</td>
</tr>
<tr>
<td>Interrupting - Conditional - Event Sub-Process - Collapsed</td>
<td><img src="image" alt="Label" /></td>
<td>SubProcess where triggeredByEvent is true and the one-and-only start event has one EventDefinition of type ConditionalEventDefinition and isInterrupting is true.</td>
<td>None or isExpanded is false</td>
</tr>
<tr>
<td>Non-interrupting - Signal - Event Sub-Process - Collapsed</td>
<td><img src="image" alt="Label" /></td>
<td>SubProcess where triggeredByEvent is true and the one-and-only start event has one EventDefinition of type SignalEventDefinition and isInterrupting is false.</td>
<td>None or isExpanded is false</td>
</tr>
<tr>
<td>Interrupting - Signal - Event Sub-Process - Collapsed</td>
<td><img src="image" alt="Label" /></td>
<td>SubProcess where triggeredByEvent is true and the one-and-only start event has one EventDefinition of type SignalEventDefinition and isInterrupting is true.</td>
<td>None or isExpanded is false</td>
</tr>
<tr>
<td>Non-interrupting - Multiple - Event Sub-Process - Collapsed</td>
<td><img src="image" alt="Label" /></td>
<td>SubProcess where triggeredByEvent is true and the one-and-only start event has multiple EventDefinitions and isInterrupting is false.</td>
<td>None or isExpanded is false</td>
</tr>
<tr>
<td>Interrupting - Multiple - Event Sub-Process - Collapsed</td>
<td><img src="image" alt="Label" /></td>
<td>SubProcess where triggeredByEvent is true and the one-and-only start event has multiple EventDefinitions and isInterrupting is true.</td>
<td>None or isExpanded is false</td>
</tr>
</tbody>
</table>
### Table 12.16 – Depiction Resolution for Collapsed Event Sub-Processes

<table>
<thead>
<tr>
<th>Non-interrupting - Parallel Multiple - Event Sub-Process - Collapsed</th>
<th><img src="Diagram.png" alt="Diagram" /></th>
<th>SubProcess where triggeredByEvent is true and the one-and-only start event has multiple EventDefinitions and isInterrupting is false and isParallelMultiple is true.</th>
<th>None or isExpanded is false</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupting - Parallel Multiple - Event Sub-Process - Collapsed</td>
<td><img src="Diagram.png" alt="Diagram" /></td>
<td>SubProcess where triggeredByEvent is true and the one-and-only start event has multiple EventDefinitions and isInterrupting is true and isParallelMultiple is true.</td>
<td>None or isExpanded is false</td>
</tr>
<tr>
<td>Non-interrupting - Escalation - Event Sub-Process - Collapsed</td>
<td><img src="Diagram.png" alt="Diagram" /></td>
<td>SubProcess where triggeredByEvent is true and the one-and-only start event has one EventDefinition of type EscalationEventDefinition and isInterrupting is false.</td>
<td>None or isExpanded is false</td>
</tr>
<tr>
<td>Interrupting - Escalation Event Sub-Process - Collapsed</td>
<td><img src="Diagram.png" alt="Diagram" /></td>
<td>SubProcess where triggeredByEvent is true and the one-and-only start event has one EventDefinition of type EscalationEventDefinition and isInterrupting is true.</td>
<td>None or isExpanded is false</td>
</tr>
<tr>
<td>Interrupting - Error - Event Sub-Process - Collapsed</td>
<td><img src="Diagram.png" alt="Diagram" /></td>
<td>SubProcess where triggeredByEvent is true and the one-and-only start event has one EventDefinition of type ErrorEventDefinition and isInterrupting is true.</td>
<td>None or isExpanded is false</td>
</tr>
<tr>
<td>Interrupting - Compensation - Event Sub-Process - Collapsed</td>
<td><img src="Diagram.png" alt="Diagram" /></td>
<td>SubProcess where triggeredByEvent is true and the one-and-only start event has one EventDefinition of type CompensationEventDefinition and isInterrupting is true.</td>
<td>None or isExpanded is false</td>
</tr>
</tbody>
</table>
### Expanded Event Sub-Processes [BPMNShape]

**Table 12.17 – Depiction Resolution for Expanded Event Sub-Processes**

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Sub-Process - Expanded</td>
<td><img src="image" alt="Label" /></td>
<td>bpmnElement: SubProcess where triggeredByEvent is true.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BPMNShape Attributes: isExpanded is true</td>
</tr>
</tbody>
</table>

### Call Activities (Calling a Global Task) [BPMNShape]

A Call Activity (Calling a Global Task) must display the Task type marker of the Global Task it calls.

Call Activities (Calling a Global Task) can also have Compensation and/or Loop Characteristic markers at the bottom center of the shape as defined above (see page 381).

**Table 12.18 – Depiction Resolution for Call Activities (Calling a Global Task)**

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Activity</td>
<td><img src="image" alt="Label" /></td>
<td>bpmnElement: CallActivity where calledElement is unspecified or of type GlobalTask.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BPMNShape Attributes: None</td>
</tr>
<tr>
<td>User Call Activity</td>
<td><img src="image" alt="Label" /></td>
<td>bpmnElement: CallActivity where calledElement is of type GlobalUserTask.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BPMNShape Attributes: None</td>
</tr>
<tr>
<td>Manual Call Activity</td>
<td><img src="image" alt="Label" /></td>
<td>bpmnElement: CallActivity where calledElement is of type GlobalManualTask.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BPMNShape Attributes: None</td>
</tr>
<tr>
<td>Business Rule Call Activity</td>
<td><img src="image" alt="Label" /></td>
<td>bpmnElement: CallActivity where calledElement is of type GlobalBusinessRuleTask.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BPMNShape Attributes: None</td>
</tr>
</tbody>
</table>
Table 12.18 – Depiction Resolution for Call Activities (Calling a Global Task)

| Script Call Activity | Call Activity where calledElement is of type GlobalScriptTask. | None |

Collapsed Call Activities (Calling a Process) [BPMNShape]

Table 12.19 – Depiction Resolution for Collapsed Call Activities (Calling a Process)

<table>
<thead>
<tr>
<th>Kind: Call Activity - Collapsed</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>bpmnElement: CallActivity where calledElement is of type Process.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BPMNShape Attributes: None or isExpanded is false</td>
</tr>
</tbody>
</table>

Expanded Call Activities (Calling a Process) [BPMNShape]

Table 12.20 – Depiction Resolution for Expanded Call Activities (Calling a Process)

<table>
<thead>
<tr>
<th>Kind: Call Activity - Expanded</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>bpmnElement: CallActivity where calledElement is of type Process.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BPMNShape Attributes: None or isExpanded is true</td>
</tr>
</tbody>
</table>

Data [BPMNShape]

Data Inputs and Data Outputs rendering are optional and only allowed for Processes.
**Table 12.21 – Depiction Resolution for Data**

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
<th>bpmnElement:</th>
<th>BPMNShape Attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Object</td>
<td>![Label]</td>
<td>DataObjectReference where dataObjectRef unspecified or is pointing to a DataObject where isCollection is false.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Data Object Collection</td>
<td>![Label]</td>
<td>DataObjectReference where dataObjectRef is pointing to a DataObject where isCollection is true.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Data Input</td>
<td>![Label]</td>
<td>DataInput where isCollection is false.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Data Input Collection</td>
<td>![Label]</td>
<td>DataInput where isCollection is true.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Data Output</td>
<td>![Label]</td>
<td>DataOutput where isCollection is false.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Data Output Collection</td>
<td>![Label]</td>
<td>DataOutput where isCollection is true.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Data Store</td>
<td>![Label]</td>
<td>DataStoreReference</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
### Events [BPMNShape]

#### Table 12.22 – Depiction Resolution for Events

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
<th>bpmnElement:</th>
<th>BPMNShape Attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>None Start Event</td>
<td></td>
<td>StartEvent with no EventDefinition</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Interrupting - Message Start Event</td>
<td></td>
<td>StartEvent with one EventDefinition of type MessageEventDefinition and isInterrupting is true.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Non-interrupting - Message Start Event</td>
<td></td>
<td>StartEvent with one EventDefinition of type MessageEventDefinition and isInterrupting is false.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Interrupting - Timer Start Event</td>
<td></td>
<td>StartEvent with one EventDefinition of type TimerEventDefinition and isInterrupting is true.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Non-interrupting - Timer Start Event</td>
<td></td>
<td>StartEvent with one EventDefinition of type TimerEventDefinition and isInterrupting is false.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Interrupting - Conditional Start Event</td>
<td></td>
<td>StartEvent with one EventDefinition of type ConditionalEventDefinition and isInterrupting is true.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Non-interrupting - Conditional Start Event</td>
<td></td>
<td>StartEvent with one EventDefinition of type ConditionalEventDefinition and isInterrupting is false.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Interrupting Signal Start Event</td>
<td></td>
<td>StartEvent One EventDefinition of type SignalEventDefinition and isInterrupting is true.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Non-interrupting - Signal Start Event</td>
<td></td>
<td>StartEvent with one EventDefinition of type SignalEventDefinition and isInterrupting is false.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Event Type</td>
<td>Description</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupting Multiple Start Event</td>
<td>StartEvent with more than one EventDefinition, parallelMultiple is false and isInterrupting is true.</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-interrupting Multiple Start Event</td>
<td>StartEvent with more than one EventDefinition, parallelMultiple is false and isInterrupting is false.</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupting - Parallel Multiple Start Event</td>
<td>StartEvent with more than one EventDefinition, parallelMultiple is true and isInterrupting is true.</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-interrupting - Parallel Multiple Start Event</td>
<td>StartEvent with more than one EventDefinition, parallelMultiple is true and isInterrupting is false.</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupting - Escalation Start Event</td>
<td>StartEvent with one EventDefinition of type EscalationEventDefinition and isInterrupting is true.</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-interrupting - Escalation Start Event</td>
<td>StartEvent with one EventDefinition of type EscalationEventDefinition and isInterrupting is false.</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupting - Error Start Event</td>
<td>StartEvent with one EventDefinition of type ErrorEventDefinition.</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupting - Compensation Start Event</td>
<td>StartEvent with one EventDefinition of type CompensationEventDefinition.</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupting - None Intermediate Event</td>
<td>IntermediateThrowEvent with no EventDefinition.</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catch - Message Intermediate Event</td>
<td>IntermediateCatchEvent with one EventDefinition of type MessageEventDefinition.</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Event Type</td>
<td>Depiction</td>
<td>Description</td>
<td>Resolution</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-----------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>Interrupting - Boundary Event</td>
<td><img src="image1" alt="Label" /></td>
<td>BoundaryEvent with one EventDefinition of type MessageEventDefinition and cancelActivity is true.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Non-interrupting Boundary Event</td>
<td><img src="image2" alt="Label" /></td>
<td>BoundaryEvent with one EventDefinition of type MessageEventDefinition and cancelActivity is false.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Throw Event</td>
<td><img src="image3" alt="Label" /></td>
<td>IntermediateThrowEvent with one EventDefinition of type MessageEventDefinition.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Timer Event</td>
<td><img src="image4" alt="Label" /></td>
<td>IntermediateCatchEvent with one EventDefinition of type TimerEventDefinition.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Interrupting - Boundary Timer Event</td>
<td><img src="image5" alt="Label" /></td>
<td>BoundaryEvent with one EventDefinition of type TimerEventDefinition and cancelActivity is true.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Non-interrupting Boundary Timer Event</td>
<td><img src="image6" alt="Label" /></td>
<td>IntermediateCatchEvent with one EventDefinition of type TimerEventDefinition and cancelActivity is false.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Conditional Event</td>
<td><img src="image7" alt="Label" /></td>
<td>IntermediateCatchEvent with one EventDefinition of type ConditionalEventDefinition.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Interrupting - Boundary Conditional Event</td>
<td><img src="image8" alt="Label" /></td>
<td>BoundaryEvent with one EventDefinition of type ConditionalEventDefinition and cancelActivity is true.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Non-interrupting Boundary Conditional Event</td>
<td><img src="image9" alt="Label" /></td>
<td>BoundaryEvent with one EventDefinition of type ConditionalEventDefinition and cancelActivity is false.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Catch Event</td>
<td><img src="image10" alt="Label" /></td>
<td>IntermediateCatchEvent with one EventDefinition of type MessageEventDefinition.</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
### Table 12.22 – Depiction Resolution for Events

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Icon</th>
<th>Description</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupting - Boundary - Catch - Signal Intermediate Event</td>
<td><img src="image" alt="Label" /></td>
<td>BoundaryEvent with one EventDefinition of type SignalEventDefinition and cancelActivity is true.</td>
<td>None</td>
</tr>
<tr>
<td>Non-interrupting-Boundary - Catch - Signal Intermediate Event</td>
<td><img src="image" alt="Label" /></td>
<td>BoundaryEvent with one EventDefinition of type SignalEventDefinition and cancelActivity is false.</td>
<td>None</td>
</tr>
<tr>
<td>Interrupting - Boundary - Throw - Signal Intermediate Event</td>
<td><img src="image" alt="Label" /></td>
<td>IntermediateThrowEvent with one EventDefinition of type SignalEventDefinition.</td>
<td>None</td>
</tr>
<tr>
<td>Catch - Multiple Intermediate Event</td>
<td><img src="image" alt="Label" /></td>
<td>IntermediateCatchEvent with more than one EventDefinition and parallelMultiple is false.</td>
<td>None</td>
</tr>
<tr>
<td>Interrupting - Boundary - Catch - Multiple Intermediate Event</td>
<td><img src="image" alt="Label" /></td>
<td>BoundaryEvent with more than one EventDefinition, parallelMultiple is false and cancelActivity is true.</td>
<td>None</td>
</tr>
<tr>
<td>Non-interrupting Boundary - Catch - Multiple Intermediate Event</td>
<td><img src="image" alt="Label" /></td>
<td>BoundaryEvent with more than one EventDefinition, parallelMultiple is false and cancelActivity is false.</td>
<td>None</td>
</tr>
<tr>
<td>Throw - Multiple Intermediate Event</td>
<td><img src="image" alt="Label" /></td>
<td>IntermediateThrowEvent with more than one EventDefinition and parallelMultiple is false.</td>
<td>None</td>
</tr>
<tr>
<td>Catch - Parallel Multiple Intermediate Event</td>
<td><img src="image" alt="Label" /></td>
<td>IntermediateCatchEvent with more than one EventDefinition and parallelMultiple is true.</td>
<td>None</td>
</tr>
<tr>
<td>Interrupting - Boundary - Catch - Parallel Multiple Intermediate Event</td>
<td><img src="image" alt="Label" /></td>
<td>BoundaryEvent with more than one EventDefinition, parallelMultiple is true and cancelActivity is true.</td>
<td>None</td>
</tr>
<tr>
<td>Non-interrupting Boundary - Catch - Parallel Multiple Intermediate Event</td>
<td><img src="image" alt="Label" /></td>
<td>BoundaryEvent with more than one EventDefinition, parallelMultiple is true and cancelActivity is false.</td>
<td>None</td>
</tr>
<tr>
<td>Event Type</td>
<td>Description</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>Interrupting - Boundary - Catch - Escalation Intermediate Event</td>
<td>BoundaryEvent with one EventDefinition of type EscalationEventDefinition and cancelActivity is true.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-interrupting - Boundary - Catch - Escalation Intermediate Event</td>
<td>BoundaryEvent with one EventDefinition of type EscalationEventDefinition and cancelActivity is false.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boundary - Catch - Error Intermediate Event</td>
<td>BoundaryEvent with one EventDefinition of type ErrorEventDefinition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boundary - Catch - Compensation Intermediate Event</td>
<td>BoundaryEvent with one EventDefinition of type CompensateEventDefinition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Throw - Compensation Intermediate Event</td>
<td>IntermediateThrowEvent with one EventDefinition of type CompensateEventDefinition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catch - Link Intermediate Event</td>
<td>IntermediateCatchEvent with one EventDefinition of type LinkEventDefinition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Throw - Link Intermediate Event</td>
<td>IntermediateThrowEvent with one EventDefinition of type LinkEventDefinition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boundary - Catch - Cancel Intermediate Event</td>
<td>BoundaryEvent with one EventDefinition of type CancelEventDefinition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>End Event Type</td>
<td>Label</td>
<td>Description</td>
<td>Resolution</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------</td>
<td>--------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>None End Event</td>
<td></td>
<td>EndEvent with no EventDefinition</td>
<td>None</td>
</tr>
<tr>
<td>Message End Event</td>
<td></td>
<td>EndEvent with one EventDefinition of type MessageEventDefinition</td>
<td>None</td>
</tr>
<tr>
<td>Signal End Event</td>
<td></td>
<td>EndEvent with one EventDefinition of type SignalEventDefinition</td>
<td>None</td>
</tr>
<tr>
<td>Multiple End Event</td>
<td></td>
<td>EndEvent with more than one EventDefinition</td>
<td>None</td>
</tr>
<tr>
<td>Escalation End Event</td>
<td></td>
<td>EndEvent with one EventDefinition of type EscalationEventDefinition</td>
<td>None</td>
</tr>
<tr>
<td>Error End Event</td>
<td></td>
<td>EndEvent with one EventDefinition of type ErrorEventDefinition</td>
<td>None</td>
</tr>
<tr>
<td>Compensation End Event</td>
<td></td>
<td>EndEvent with one EventDefinition of type CompensateEventDefinition</td>
<td>None</td>
</tr>
<tr>
<td>Cancel End Event</td>
<td></td>
<td>EndEvent with one EventDefinition of type CancelEventDefinition</td>
<td>None</td>
</tr>
<tr>
<td>Terminate End Event</td>
<td></td>
<td>EndEvent with one EventDefinition of type TerminateEventDefinition</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 12.22 – Depiction Resolution for Events
## Gateways [BPMNShape]

### Table 12.23 – Depiction Resolution for Gateways

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
<th>bpmnElement:</th>
<th>BPMNShape Attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exclusive Gateway -</td>
<td></td>
<td></td>
<td>ExclusiveGateway</td>
<td>None or isMarkerVisible is false</td>
</tr>
<tr>
<td>without Marker</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exclusive Gateway -</td>
<td></td>
<td></td>
<td>ExclusiveGateway</td>
<td>isMarkerVisible is true</td>
</tr>
<tr>
<td>with Marker</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inclusive Gateway</td>
<td></td>
<td></td>
<td>InclusiveGateway</td>
<td>None</td>
</tr>
<tr>
<td>Parallel Gateway</td>
<td></td>
<td></td>
<td>ParallelGateway</td>
<td>None</td>
</tr>
<tr>
<td>Complex Gateway</td>
<td></td>
<td></td>
<td>ComplexGateway</td>
<td>None</td>
</tr>
<tr>
<td>Event-Based Gateway</td>
<td></td>
<td></td>
<td>EventBasedGateway where instantiate is false.</td>
<td>None</td>
</tr>
<tr>
<td>Event-Based Gateway to</td>
<td></td>
<td></td>
<td>EventBasedGateway where instantiate is true and eventGatewayType is exclusive.</td>
<td>None</td>
</tr>
<tr>
<td>Start a Process</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 12.23 – Depiction Resolution for Gateways

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>![Group Image]</td>
<td><strong>bpmnElement:</strong> Group <strong>Attributes:</strong> None</td>
</tr>
<tr>
<td>Text Annotation</td>
<td>![Text Annotation Image]</td>
<td><strong>bpmnElement:</strong> Text Annotation <strong>Attributes:</strong> None</td>
</tr>
</tbody>
</table>

**Parallel Event-Based Gateway to Start a Process**

EventBasedGateway where instantiate is true and eventGatewayType is parallel.

### Table 12.24 – Depiction Resolution for Artifacts

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>![Group Image]</td>
<td><strong>bpmnElement:</strong> Group <strong>Attributes:</strong> None</td>
</tr>
<tr>
<td>Text Annotation</td>
<td>![Text Annotation Image]</td>
<td><strong>bpmnElement:</strong> Text Annotation <strong>Attributes:</strong> None</td>
</tr>
</tbody>
</table>

### Table 12.25 – Depiction Resolution for Lanes

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal Lane</td>
<td>![Horizontal Lane Image]</td>
<td><strong>bpmnElement:</strong> Lane <strong>Attributes:</strong> None or isVertical is false</td>
</tr>
<tr>
<td>Vertical Lane</td>
<td>![Vertical Lane Image]</td>
<td><strong>bpmnElement:</strong> Lane <strong>Attributes:</strong> isVertical is true</td>
</tr>
</tbody>
</table>
Pools [BPMNShape]

Table 12.26 – Depiction Resolution for Pools

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>bpmnElement: Participant where ParticipantMultiplicity is unspecified or set and its maximum attribute is 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BPMNShape Attributes: None or isVertical is false</td>
</tr>
<tr>
<td>Horizontal Pool</td>
<td><img src="image" alt="Horizontal Pool" /></td>
<td>Participant where ParticipantMultiplicity is set and its maximum attribute is &gt; 1.</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Horizontal Pool" /></td>
<td>None or isVertical is false</td>
</tr>
<tr>
<td>Horizontal Pool - with Multi Instance Participant</td>
<td><img src="image" alt="Horizontal Pool" /></td>
<td>Participant where ParticipantMultiplicity is set and its maximum attribute is &gt; 1.</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Horizontal Pool" /></td>
<td>None or isVertical is false</td>
</tr>
<tr>
<td>Vertical Pool</td>
<td><img src="image" alt="Vertical Pool" /></td>
<td>Participant where ParticipantMultiplicity is unspecified or set and its maximum attribute is 1.</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Vertical Pool" /></td>
<td>isVertical is true</td>
</tr>
<tr>
<td>Vertical Pool - with Multi Instance Participant</td>
<td><img src="image" alt="Vertical Pool" /></td>
<td>Participant where ParticipantMultiplicity is set and its maximum attribute is &gt; 1.</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Vertical Pool" /></td>
<td>isVertical is true</td>
</tr>
</tbody>
</table>

Choreography Tasks [BPMNShape]

While the depictions provided in Table 12.27 - Depiction Resolution for Choreography Tasks contain Participant Bands, Participant Bands are separate shapes that need to be separately defined. Individual Participant Bands are rendered by separate BPMNShape(s), each Participant Band referencing the corresponding participant. See page 407.
Table 12.27 – Depiction Resolution for Choreography Tasks

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choreography Task</td>
<td></td>
<td>ChoreographyTask where loopType is None.</td>
</tr>
<tr>
<td>Choreography Task - Loop</td>
<td></td>
<td>ChoreographyTask where loopType is Standard.</td>
</tr>
<tr>
<td>Choreography Task - Sequential Multi Instance</td>
<td></td>
<td>ChoreographyTask where loopType is MultiInstanceSequential.</td>
</tr>
<tr>
<td>Choreography Task - Parallel Multi Instance</td>
<td></td>
<td>ChoreographyTask where loopType is MultiInstanceParallel.</td>
</tr>
</tbody>
</table>

**Collapsed Sub-Choreographies [BPMNShape]**

While the depictions provided in Table 12.28 - Depiction Resolution for Collapsed Sub-Choreographies contain Participant Bands, Participant Bands are separate shapes that need to be separately defined. Individual Participant Bands are rendered by separate BPMNShape(s), each Participant Band referencing the corresponding participant. See page 407.
Table 12.28 – Depiction Resolution for Sub-Choreographies (Collapsed)

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
<th>bpmnElement:</th>
<th>BPMNShape Attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub-Choreography - Collapsed</td>
<td></td>
<td>SubChoreography where loopType is None.</td>
<td>None or isExpanded is false</td>
<td></td>
</tr>
<tr>
<td>Sub-Choreography - Loop - Collapsed</td>
<td></td>
<td>SubChoreography where loopType is Standard.</td>
<td>None or isExpanded is false</td>
<td></td>
</tr>
<tr>
<td>Sub-Choreography - Sequential Multi Instance - Collapsed</td>
<td></td>
<td>SubChoreography where loopType is MultiInstanceSequential.</td>
<td>None or isExpanded is false</td>
<td></td>
</tr>
<tr>
<td>Sub-Choreography - Parallel Multi Instance - Collapsed</td>
<td></td>
<td>SubChoreography where loopType is MultiInstanceParallel.</td>
<td>None or isExpanded is false</td>
<td></td>
</tr>
</tbody>
</table>

**Expanded Sub-Choreographies [BPMNShape]**

While the depiction provided in Table 12.29 - Depiction Resolution for Expanded Sub-Choreographies contains Participant Bands, Participant Bands are separate shapes that need to be separately defined. Individual Participant Bands are rendered by separate BPMNShape(s), each Participant Band referencing the corresponding participant. See page 407.

An expanded Sub Choreography has a loop type that is depicted exactly like the collapsed version in Table 12.28 - Depiction Resolution for Collapsed Sub-Choreographies above.
Call Choreographies (Calling a Global Choreography Task) [BPMNShape]

While the depictions provided in Table 12.30 - Depiction Resolution for Call Choreographies (Calling a Global Choreography Task) contain Participant Bands, Participant Bands are separate shapes that need to be separately defined. Individual Participant Bands are rendered by separate BPMNShape(s), each Participant Band referencing the corresponding participant. See page 407.

Table 12.29 – Depiction Resolution for Sub-Choreographies (Expanded)

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub-Choreography - Expanded</td>
<td>![Image]</td>
<td>bpmnElement: SubChoreography Attributes: isExpanded is true</td>
</tr>
</tbody>
</table>

Table 12.30 – Depiction Resolution for Call Choreographies (Calling a Global Choreography Task)

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Choreography Activity calling a Global Choreography Task</td>
<td>![Image]</td>
<td>CallChoreography where calledChoreographyRef is unspecified or of type GlobalChoreographyTask and loopType is None. Attributes: None</td>
</tr>
<tr>
<td>Call Choreography Activity calling a Global Choreography Task - Loop</td>
<td>![Image]</td>
<td>CallChoreography where calledChoreographyRef is of type GlobalChoreographyTask and loopType is Standard. Attributes: None</td>
</tr>
</tbody>
</table>
Collapsed Call Choreographies (Calling a Choreography) [BPMNShape]

While the depictions provided in Table 12.31 - Depiction Resolution for Collapsed Call Choreography (Calling a Choreography) contain Participant Bands, Participant Bands are separate shapes that need to be separately defined. Individual Participant Bands are rendered by separate BPMNShape(s), each Participant Band referencing the corresponding participant. See page 407.

Table 12.31 – Depiction Resolution for Collapsed Call Choreographies (Calling a Choreography)

<table>
<thead>
<tr>
<th>Kind: Call Choreography Activity calling a Choreography - Loop</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kind:</strong> Call Choreography Activity calling a Choreography</td>
<td><strong>Depiction:</strong></td>
<td><strong>Specific Depiction Resolution:</strong></td>
</tr>
<tr>
<td><strong>Attributes:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallChoreography where calledChoreographyRef is of type Choreography and loopType is Standard.</td>
<td>None or isExpanded is false</td>
<td>None or isExpanded is false</td>
</tr>
<tr>
<td>CallChoreography where calledChoreographyRef is of type Choreography and loopType is Standard.</td>
<td>None or isExpanded is false</td>
<td>None or isExpanded is false</td>
</tr>
<tr>
<td>CallChoreography where calledChoreographyRef is of type Choreography and loopType is Standard.</td>
<td>None or isExpanded is false</td>
<td>None or isExpanded is false</td>
</tr>
</tbody>
</table>

Table 12.30 – Depiction Resolution for Call Choreographies (Calling a Global Choreography Task)

<table>
<thead>
<tr>
<th>Kind: Call Choreography Activity calling a Choreography - Loop</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kind:</strong> Call Choreography Activity calling a Choreography</td>
<td><strong>Depiction:</strong></td>
<td><strong>Specific Depiction Resolution:</strong></td>
</tr>
<tr>
<td><strong>Attributes:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallChoreography where calledChoreographyRef is of type GlobalChoreographyTask and loopType is MultiInstanceSequential.</td>
<td>None or isExpanded is false</td>
<td>None or isExpanded is false</td>
</tr>
</tbody>
</table>
Expanded Call Choreographies (Calling a Choreography) [BPMNShape]

While the depiction provided in Table 12.32 - Depiction Resolution for Expanded Call Choreographies (Calling a Choreography) contains Participant Bands, Participant Bands are separate shapes that need to be separately defined. Individual Participant Bands are rendered by separate BPMNShape(s), each Participant Band referencing the corresponding participant. See page 407.

An expanded Use Sub Choreography has a loop type that is depicted exactly like the collapsed version in Table 12.31 - Depiction Resolution for Collapsed CallChoreography (Calling a Choreography) above.

Table 12.32 – Depiction Resolution for Expanded Call Choreographies (Calling a Choreography)

<table>
<thead>
<tr>
<th>Kind: Call Choreography Activity calling a Choreography</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallChoreography where calledChoreographyRef is of type Choreography</td>
<td></td>
<td>isExpanded is true.</td>
</tr>
<tr>
<td>CallChoreography where calledChoreographyRef is of type Choreography and loopType is MultiInstanceParallel.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Choreography Participant Bands [BPMNShape]

Participant Bands (used in Choreography shapes) are separate shapes that need to be separately defined. Individual Participant Bands are rendered by separate BPMNShape. Each Participant Band referencing the corresponding participant.

Note that for Participant Bands with the envelope decorator, the envelope decorator should be depicted close to the band, vertically centered with the band, and linked to the band using a dotted line. The name of the message may be used as a label for the envelop decorator. BPMN DI does not provide an interchange of the bounds of the label of the envelope decorator.
The bounds of the BPMNShape representing the band do not include the envelope decorator. The envelope decorator is therefore outside of the BPMNShape bounds. BPMN DI does not provide an interchange of the bounds of the envelope decorator.

### Table 12.33 – Depiction Resolution for Choreography Participant Bands

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>bpmnElement: BPMNShape Attributes:</td>
</tr>
<tr>
<td>Initiating Participant - Top</td>
<td><img src="image" alt="Label" /></td>
<td>Participant where participantMultiplicity is unspecified or is set and its maximum attribute is 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>participantBandKind is top_initiating and isMessageVisible is unspecified or false.</td>
</tr>
<tr>
<td>Initiating Participant - Top with Decorator</td>
<td><img src="image" alt="Envelope" /> <img src="image" alt="Label" /></td>
<td>Participant where participantMultiplicity is unspecified or set and its maximum attribute is 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>participantBandKind is top_initiating and isMessageVisible is true.</td>
</tr>
<tr>
<td>Initiating - Additional Participant</td>
<td><img src="image" alt="Label" /></td>
<td>Participant where participantMultiplicity is unspecified or set and its maximum attribute is 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>participantBandKind is middle_initiating.</td>
</tr>
<tr>
<td>Initiating Participant - Bottom</td>
<td><img src="image" alt="Label" /></td>
<td>Participant where participantMultiplicity is unspecified or set and its maximum attribute is 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>participantBandKind is bottom_initiating and isMessageVisible is unspecified or false.</td>
</tr>
<tr>
<td>Initiating Participant - Bottom with Decorator</td>
<td><img src="image" alt="Envelope" /> <img src="image" alt="Label" /></td>
<td>Participant where participantMultiplicity is unspecified or set and its maximum attribute is 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>participantBandKind is bottom_initiating and isMessageVisible is true.</td>
</tr>
<tr>
<td>Initiating - Top - Multi-Instance Participant</td>
<td><img src="image" alt="Label III" /></td>
<td>Participant where participantMultiplicity is unspecified or set and its maximum attribute is &gt; 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>participantBandKind is top_initiating and isMessageVisible is unspecified or false.</td>
</tr>
<tr>
<td>Initiating - Top - Multi-Instance Participant with Decorator</td>
<td><img src="image" alt="Envelope" /> <img src="image" alt="Label III" /></td>
<td>Participant where participantMultiplicity is unspecified or set and its maximum attribute is &gt; 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>participantBandKind is top_initiating and isMessageVisible is true.</td>
</tr>
<tr>
<td>Initiating - Additional Multi-Instance Participant</td>
<td><img src="image" alt="Label III" /></td>
<td>Participant where participantMultiplicity is unspecified or set and its maximum attribute is &gt; 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>participantBandKind is middle_initiating.</td>
</tr>
</tbody>
</table>
### Table 12.33 – Depiction Resolution for Choreography Participant Bands

<table>
<thead>
<tr>
<th>Participant Type</th>
<th>Participant Description</th>
<th>ParticipantBandKind</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiating - Bottom - Multi-Instance Participant</td>
<td>Participant where (\text{participantMultiplicity}) is unspecified or set and its maximum attribute is (&gt; 1).</td>
<td>participantBandKind is bottom_initiating and isMessageVisible is unspecified or false.</td>
</tr>
<tr>
<td>Initiating - Bottom - Multi-Instance Participant with Decorator</td>
<td>Participant where (\text{participantMultiplicity}) is unspecified or set and its maximum attribute is (&gt; 1).</td>
<td>participantBandKind is bottom_initiating and isMessageVisible is true.</td>
</tr>
<tr>
<td>Non Initiating - Top Participant</td>
<td>Participant where (\text{participantMultiplicity}) is set and its maximum attribute is (= 1).</td>
<td>participantBandKind is top_non_initiating and isMessageVisible is unspecified or false.</td>
</tr>
<tr>
<td>Non Initiating - Top Participant with Decorator</td>
<td>Participant where (\text{participantMultiplicity}) is set and its maximum attribute is (= 1).</td>
<td>participantBandKind is top_non_initiating and isMessageVisible is true.</td>
</tr>
<tr>
<td>Non Initiating - Additional Participant</td>
<td>Participant where (\text{participantMultiplicity}) is set and its maximum attribute is (= 1).</td>
<td>participantBandKind is middle_non_initiating.</td>
</tr>
<tr>
<td>Non Initiating - Bottom Participant</td>
<td>Participant where (\text{participantMultiplicity}) is set and its maximum attribute is (= 1).</td>
<td>participantBandKind is bottom_non_initiating and isMessageVisible is unspecified or false.</td>
</tr>
<tr>
<td>Non Initiating - Bottom Participant with Decorator</td>
<td>Participant where (\text{participantMultiplicity}) is set and its maximum attribute is (= 1).</td>
<td>participantBandKind is bottom_non_initiating and isMessageVisible is true.</td>
</tr>
<tr>
<td>Non Initiating - Top - Multi-Instance Participant</td>
<td>Participant where (\text{participantMultiplicity}) is set and its maximum attribute is (&gt; 1).</td>
<td>participantBandKind is top_non_initiating and isMessageVisible is unspecified or false.</td>
</tr>
<tr>
<td>Non Initiating - Top - Multi-Instance Participant with Decorator</td>
<td>Participant where (\text{participantMultiplicity}) is set and its maximum attribute is (&gt; 1).</td>
<td>participantBandKind is top_non_initiating and isMessageVisible is true.</td>
</tr>
<tr>
<td>Non Initiating - Additional Multi-Instance Participant</td>
<td>Participant where (\text{participantMultiplicity}) is set and its maximum attribute is (&gt; 1).</td>
<td>participantBandKind is middle_non_initiating.</td>
</tr>
</tbody>
</table>
Table 12.33 – Depiction Resolution for Choreography Participant Bands

<table>
<thead>
<tr>
<th>Kind: Non Initiating - Bottom - Multi-Instance Participant</th>
<th>Depiction: Participant where ParticipantMultiplicity is set and its maximum attribute is &gt; 1.</th>
<th>Attributes: participantBandKind is bottom_non_initiating and isMessageVisible is unspecified or false.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind: Non Initiating - Bottom - Multi-Instance Participant with Decorator</td>
<td>Depiction: Participant where participantMultiplicity is set and its maximum attribute is &gt; 1.</td>
<td>Attributes: participantBandKind is bottom_non_initiating and isMessageVisible is true.</td>
</tr>
</tbody>
</table>

Conversations [BPMNShape]

Table 12.34 – Depiction Resolution for Conversations

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conversation</td>
<td><img src="image" alt="Conversation" /></td>
<td>Conversation where calledCollaborationRef is a GlobalConversation.</td>
</tr>
<tr>
<td>Sub-Conversation</td>
<td><img src="image" alt="Sub-Conversation" /></td>
<td>SubConversation</td>
</tr>
<tr>
<td>Call Conversation</td>
<td><img src="image" alt="Call Conversation" /></td>
<td>CallConversation where calledCollaborationRef is a Collaboration.</td>
</tr>
<tr>
<td>Call Conversation</td>
<td><img src="image" alt="Call Conversation" /></td>
<td>CallConversation where calledCollaborationRef is a Collaboration.</td>
</tr>
</tbody>
</table>
12.3.3 BPMNEdge

Connecting Objects [BPMNEdge]

The target [targetElement] and source [sourceElement] of a BPMNEdge may be redefined when the depiction of the source or target of the edge is different than the target [targetRef] and source [sourceRef] of the referenced model element [bpmnElement] (e.g., Message flow finishing on the border of a black box Pool or a collapsed Sub-Process rather than the actual Flow Node within the Pool or Sub-Process). In such case, the targetElement and/or sourceElement of the BPMNEdge must point to the appropriate BPMNShape or BPMNEdge.

The source [sourceElement] and target [targetElement] of a BPMNEdge can never be a BPMNShape with participantBandKind set (i.e., only Choreography Activity can be source or target of the BPMNEdge not the Participant Bands).

Note that for Message Flow with an envelope decorator, the envelope decorator should be at the midpoint of the message flow. BPMN DI does not provide an interchange of the bounds of the envelope decorator.

The “diamond” at the source of the Conditional Sequence Flow should not be depicted when the source of a Conditional Sequence Flow is a Gateway. In other words, when the source of a Conditional Sequence Flow is a Gateway, the Conditional Sequence Flow looks like a Sequence Flow.

Even though DataInputAssociation(s) and DataOutputAssociation(s) (Directed Data Associations) always point to DataInput(s) or DataOutput(s) as sources or targets within the BPMN model, they are mostly depicted as starting or finishing on the border of a different depicted element and thus, the target [targetElement] or source [sourceElement] of the BPMNEdge must be specified.

Table 12.35 – Depiction Resolution for Connecting Objects

<table>
<thead>
<tr>
<th>Kind:</th>
<th>Depiction:</th>
<th>Specific Depiction Resolution:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>bpmnElement: SequenceFlow where default is false and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>conditionExpression is unspecified.</td>
</tr>
<tr>
<td>Sequence Flow</td>
<td><img src="image" alt="Label" /></td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td><img src="image" alt="Label" /></td>
<td>SequenceFlow where default is false and conditionExpression is unspecified.</td>
</tr>
<tr>
<td>Sequence Flow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default Sequence Flow</td>
<td><img src="image" alt="Label" /></td>
<td>SequenceFlow where default is true and conditionExpression is unspecified.</td>
</tr>
<tr>
<td>Message Flow</td>
<td><img src="image" alt="Label" /></td>
<td>MessageFlow</td>
</tr>
<tr>
<td>Initiating Message Flow</td>
<td><img src="image" alt="Label" /></td>
<td>MessageFlow</td>
</tr>
</tbody>
</table>

Business Process Model and Notation, v2.0 411
12.4 Example(s)

This section provides examples to support interpretation of the BPMN DI specification. Some BPMN diagram depictions along with their XML BPMN DI serializations are provided. The XML samples provided in this section present only BPMN DI instances and omit the BPMN 2.0 abstract syntax.

For readability purposes, the bpmnElement that is referenced by the BPMNPlane, BPMNShape and BPMNEdge use a representative string pattern. This string pattern is:

```
BPMNModelClassName_BPMNModelNameAttributeValue
```

For example: “Task_Activity” for a Task named “Activity.”

In the provided XML serializations, the di namespace refers to the Diagram Interchange namespace defined in Annex B, and the dc namespace refers to the Diagram Common namespace also defined in Annex B.

### 12.4.1 Depicting Content in a Sub-Process

This section shows various ways of depicting the content of a Sub-Process of the same BPMN model.

The BPMN model contains a process composed of a none start event (named “StartEvent”), a sub-process (named “SubProcess”) and a none end event (named “EndEvent”). There is a sequence flow (named “a”) between the start event (named “StartEvent”) and the sub-process (named “SubProcess”) and a sequence flow (named “d”) between the sub-process (named “SubProcess”) and the end event (named “EndEvent”).
The sub-process (named “SubProcess”) is composed of a none start event (named “SubProcessStart”), an abstract task (named “Activity”) and a none end event (named “SubProcessEnd”). There is a sequence flow (named “b”) between the start event (named “SubProcessStart”) and the task (named “Activity”) and a sequence flow (named “c”) between the task (named “Activity”) and the end event (named “SubProcessEnd”).

**Expanded Sub-Process**

First, a BPMN diagram depicts the BPMN model with the expanded sub-process showing its content (see Figure 12.8). This leads to a BPMN DI serialization of a single diagram that depicts this process (see Table 12.37 - Expanded Sub-Process BPMN DI instance).

![Figure 12.8 - Expanded Sub-Process Example](image)

**Table 12.36 - Expanded Sub-Process BPMN DI instance**

```
<BPMNDiagram name="Events Inside the Sub Process" resolution="72">
  <BPMNPlane bpmnElement="Process_Process">
    <BPMNShape bpmnElement="StartEvent_StartEvent">
      <dc:Bounds height="30.0" width="30.0" x="120.0" y="225.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="SubProcess_SubProcess" isExpanded="true">
      <dc:Bounds height="168.0" width="348.0" x="192.0" y="156.0" id="BorderSubProcess"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="StartEvent_SubProcessStart" id="BorderStart">
      <dc:Bounds height="30.0" width="30.0" x="228.0" y="225.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="Task_Activity">
      <dc:Bounds height="68.0" width="83.0" x="324.0" y="206.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="EndEvent_SubProcessEnd">
      <dc:Bounds height="32.0" width="32.0" x="468.0" y="224.0" id="BorderEnd"/>
      <BPMNLabel/>
    </BPMNShape>
  </BPMNPlane>
</BPMNDiagram>
```
Expanded Sub-Process with Start and End Events on Border

An alternative to depicting the same BPMN model of Section 12.4.1 would be to place the sub-process start and end events on the border of the sub-process (see Figure 12.9). In the BPMN DI serialization of this diagram (see Table 12.40 - Expanded Sub-Process with Start and End Events on Border BPMN DI instance), the target of the sequence flow named “a” and the source of the sequence flow named “d” are the start and end events on the boundary of the sub-process.

Compare the target of the sequence flow named “a” and the source of the sequence flow named “d” of Table 12.37 - Expanded Sub-Process BPMN DI instance with that of Table 12.38 - Start and End Events on the Border BPMN DI instance.
Figure 12.9 - Start and End Events on the Border Example

Table 12.37 - Start and End Events on the Border BPMN DI instance

```xml
<BPMNDiagram name="StartAndEdnEventsOnTheBorder" resolution="72">
  <BPMNPlane bpmnElement="Process_Process">
    <BPMNShape bpmnElement="StartEvent_StartEvent">
      <dc:Bounds height="30.0" width="30.0" x="120.0" y="225.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="SubProcess_SubProcess" isExpanded="true">
      <dc:Bounds height="168.0" width="348.0" x="192.0" y="156.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="StartEvent_SubProcessStart">
      <dc:Bounds height="30.0" width="30.0" x="177.0" y="225.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="Task_Activity">
      <dc:Bounds height="68.0" width="83.0" x="324.0" y="206.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="EndEvent_SubProcessEnd">
      <dc:Bounds height="32.0" width="32.0" x="524.0" y="224.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="EndEvent_EndEvent">
      <dc:Bounds height="32.0" width="32.0" x="604.0" y="224.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNEdge bpmnElement="SequenceFlow_a">
      <di:waypoint x="150.0" y="240.0"/>
      <di:waypoint x="177.0" y="240.0"/>
      <BPMNLabel/>
    </BPMNEdge>
    <BPMNEdge bpmnElement="SequenceFlow_b">
      <di:waypoint x="207.0" y="240.0"/>
      <di:waypoint x="324.0" y="240.0"/>
      <BPMNLabel/>
    </BPMNEdge>
  </BPMNPlane>
</BPMNDiagram>
```
Collapsed Sub-Process

Alternatively, one could depict the same BPMN model of Section 12.4.1 as two diagrams. A first diagram (Figure 12.10) depicts the process with the sub-process collapsed, while a second diagram (Figure 12.11) depicts the content of the sub-process.

Figure 12.10 - Collapsed Sub-Process

Figure 12.11 - Contents of Collapsed Sub-Process

Table 12.38 - Collapsed Sub-Process BPMN DI instance
Table 12.39 - Sub-Process Content BPMN DI instance

```xml
<BPMNDiagram name="SubProcess" resolution="72">
  <BPMNPlane bpmnElement="SubProcess_SubProcess">
    <BPMNShape bpmnElement="StartEvent_SubProcessStart">
      <dc:Bounds height="30.0" width="30.0" x="208.0" y="219.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="Task_Activity">
      <dc:Bounds height="68.0" width="83.0" x="304.0" y="200.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="EndEvent_SubProcessEnd">
      <dc:Bounds height="32.0" width="32.0" x="448.0" y="218.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNEdge bpmnElement="SequenceFlow_b">
      <di:waypoint x="238.0" y="234.0"/>
      <di:waypoint x="304.0" y="234.0"/>
      <BPMNLabel/>
    </BPMNEdge>
    <BPMNEdge bpmnElement="SequenceFlow_c">
      <di:waypoint x="387.0" y="234.0"/>
      <di:waypoint x="448.0" y="234.0"/>
      <BPMNLabel/>
    </BPMNEdge>
  </BPMNPlane>
</BPMNDiagram>
```
12.4.2 Multiple Lanes and Nested Lanes

In this next example, a diagram depicting a BPMN Process is composed of a LaneSet which contains 2 lanes is presented. The second lane contains 2 sub lanes (See Figure 12.12).

![Diagram of Multiple Lanes and Nested Lanes](image)

**Figure 12.12 - Nested Lanes Example**

**Table 12.40 - Multiple Lanes and Nested Lanes BPMN DI instance**

```xml
<BPMNDiagram name="Lanes and Nested Lanes" resolution="72">
  <BPMNPlane bpmnElement="Process_LanesAndNestedLanes">
    <BPMNShape bpmnElement="Lane_Lane1" isHorizontal="true">
      <dc:Bounds height="144.0" width="498.0" x="87.0" y="144.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="Lane_Lane2" isHorizontal="true">
      <dc:Bounds height="162.0" width="498.0" x="87.0" y="288.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="Lane_Lane2_2" isHorizontal="true">
      <dc:Bounds height="78.0" width="474.0" x="111.0" y="372.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="Lane_Lane2_1" isHorizontal="true">
      <dc:Bounds height="84.0" width="474.0" x="111.0" y="288.0"/>
      <BPMNLabel/>
    </BPMNShape>
  </BPMNPlane>
</BPMNDiagram>
```
12.4.3 Vertical Collaboration

In this example, a Collaboration between two Participants (Pool A and Pool B) is depicted. The first Participant is depicted with a white box Pool and the second Participant is depicted with a black box Pool. This diagram also depicts message flows that are decorated with message envelopes (See Figure 12.13).

![Figure 12.13 - Vertical Collaboration Example](image)

### Table 12.41 - Vertical Collaboration BPMN DI instance

```xml
<BPMNDiagram name="Vertical Collaboration" resolution="72">
  <BPMNPlane bpmnElement="Collaboration_Vertical_Collaboration">
    <BPMNShape bpmnElement="Participant_Pool_A" isHorizontal="false">
      <dc:Bounds height="258.0" width="336.0" x="96.0" y="276.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="Lane_Lane1" isHorizontal="false">
      <dc:Bounds height="228.0" width="168.0" x="96.0" y="306.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="Lane_Lane2" isHorizontal="false">
      <dc:Bounds height="228.0" width="168.0" x="264.0" y="306.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="Participant_Pool_B" isHorizontal="false">
      <dc:Bounds height="258.0" width="336.0" x="624.0" y="279.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="TaskReceiving_Receiving">
      <dc:Bounds height="68.0" width="83.0" x="145.0" y="436.0"/>
      <BPMNLabel/>
    </BPMNShape>
    <BPMNShape bpmnElement="TaskSending_Sending">
      <dc:Bounds height="68.0" width="83.0" x="282.0" y="338.0"/>
      <BPMNLabel/>
    </BPMNShape>
  </BPMNPlane>
</BPMNDiagram>
```
12.4.4 Conversation

The following diagram depicts a Collaboration between 3 Participants (Participants 1, 2 and 3) including two Conversations. The diagram also has an annotation connected to a message flow (See Figure 12.14).

![Conversation Diagram](image)

**Figure 12.14 - Conversation Example**

**Table 12.42 - Conversation BPMN DI instance**

```xml
<bpmndi:BPMNDiagram name="Conversation" resolution="72">
  <bpmndi:BPMNPlane bpmnElement="Collaboration_Conversation">
    <Participant1>
      
    </Participant1>
    
    <Participant2>
      
    </Participant2>
    
    <Participant3>
      
    </Participant3>
    
    <Conversation1>
      
    </Conversation1>
    
    <Conversation2>
      
    </Conversation2>
    
    <MessageAnnotation>
      
    </MessageAnnotation>
  </BPMNPlane>
</bpmndi:BPMNDiagram>
```
12.4.5 Choreography

The following diagram depicts a Choreography consisting of 3 Choreography Activities (2 Choreography Tasks and 1 SubChoreography). This diagram also depicts Participant Bands with and without envelope decorator.

![Choreography Diagram](image)

Figure 12.15 - Choreography Example

Table 12.43 - Choreography BPMN DI instance

```xml
<di:waypoint x="181.0" y="360.0"/>
<di:waypoint x="181.0" y="324.0"/>
<di:waypoint x="210.0" y="324.0"/>
<bpmndi:BPMNLabel/>
</bpmndi:BPMNEdge>
</bpmndi:BPMNPlane>
</bpmndi:BPMNDiagram>
```
</bpmndi:BPMNEdge>
<bpmndi:BPMNEdge bpmnElement="SequenceFlow_b">
  <di:waypoint x="252.0" y="153.0"/>
  <di:waypoint x="312.0" y="153.0"/>
  <bpmndi:BPMNLabel/>
</bpmndi:BPMNEdge>
<bpmndi:BPMNEdge bpmnElement="SequenceFlow_c">
  <di:waypoint x="408.0" y="153.0"/>
  <di:waypoint x="468.0" y="153.0"/>
  <bpmndi:BPMNLabel/>
</bpmndi:BPMNEdge>
<bpmndi:BPMNEdge bpmnElement="SequenceFlow_d">
  <di:waypoint x="564.0" y="153.0"/>
  <di:waypoint x="624.0" y="153.0"/>
  <bpmndi:BPMNLabel/>
</bpmndi:BPMNEdge>
</bpmndi:BPMNPlane>
</bpmndi:BPMNDiagram>
13 BPMN Execution Semantics

Note – The content of this chapter is REQUIRED for BPMN Process Execution Conformance or for BPMN Complete Conformance. However, this chapter 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 2.

This section defines the execution semantics for orchestrations in BPMN 2.0. 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 section 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
13.1 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.

13.2 Activities

This section 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.
13.2.1 Sequence Flow Considerations

The nature and behavior of Sequence Flows is described in “Sequence Flow” on page 97. 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.

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.
13.2.2 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](image-url)
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 Section 10.3.2 (page 225) 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.
13.2.3 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.

13.2.4 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*.

### 13.2.5 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.

13.2.6 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.

13.2.7 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 cancelled, 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
13.3 Gateways

This section describes the behavior of Gateways.

13.3.1 Parallel Gateway (Fork and Join)

![Figure 13.3 - Merging and Branching Sequence Flows for a Parallel Gateway](image)

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>
<thead>
<tr>
<th>Operational Semantics</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception Issues</td>
<td>The Parallel Gateway cannot throw any exception.</td>
</tr>
</tbody>
</table>
| Workflow Patterns Support | Parallel Split (WCP-2)  
Synchronization (WCP-3) |
13.3.2 Exclusive Gateway (Exclusive Decision (data-based) and Exclusive Merge)

![Diagram of Exclusive Gateway](image)

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>
<thead>
<tr>
<th>Table 13.2 – Exclusive Gateway Execution Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operational Semantics</strong></td>
</tr>
<tr>
<td>Each <em>token</em> arriving at any incoming <em>Sequence Flows</em> activates the gateway and is routed to exactly one of the outgoing <em>Sequence Flows</em>. In order to determine the outgoing <em>Sequence Flows</em> that receives the <em>token</em>, the conditions are evaluated in order. The first condition that evaluates to true determines the <em>Sequence Flow</em> the <em>token</em> is sent to. No more conditions are henceforth evaluated. If and only if none of the conditions evaluates to true, the <em>token</em> is passed on the default <em>Sequence Flow</em>. In case all conditions evaluate to false and a default flow has not been specified, an exception is thrown.</td>
</tr>
<tr>
<td><strong>Exception Issues</strong></td>
</tr>
<tr>
<td>The exclusive gateway throws an exception in case all conditions evaluate to false and a default flow has not been specified.</td>
</tr>
<tr>
<td><strong>Workflow Patterns Support</strong></td>
</tr>
<tr>
<td>Exclusive Choice (WCP-4)</td>
</tr>
<tr>
<td>Simple Merge (WCP-5)</td>
</tr>
<tr>
<td>Multi-Merge (WCP-8)</td>
</tr>
</tbody>
</table>

13.3.3 Inclusive Gateway (Inclusive Decision and Inclusive Merge)

![Diagram of Inclusive Gateway](image)

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>
<thead>
<tr>
<th>Operational Semantics</th>
<th>The <strong>Inclusive Gateway</strong> is activated if</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• At least one incoming <strong>Sequence Flow</strong> has at least one <em>token</em> and</td>
</tr>
<tr>
<td></td>
<td>• For every directed path formed by sequence flow that</td>
</tr>
<tr>
<td></td>
<td>- starts with a <strong>Sequence Flow</strong> ( f ) of the diagram that has a <em>token</em>,</td>
</tr>
<tr>
<td></td>
<td>- ends with an <em>incoming Sequence Flow</em> of the inclusive gateway that has <em>no token</em>, and</td>
</tr>
<tr>
<td></td>
<td>- does not visit the <strong>Inclusive Gateway</strong>.</td>
</tr>
<tr>
<td></td>
<td>• There is also a directed path formed by <strong>Sequence Flow</strong> that</td>
</tr>
<tr>
<td></td>
<td>- starts with ( f ),</td>
</tr>
<tr>
<td></td>
<td>- ends with an <em>incoming Sequence Flow</em> of the inclusive gateway that has a <em>token</em>, and</td>
</tr>
<tr>
<td></td>
<td>- does not visit the <strong>Inclusive Gateway</strong>.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Exception Issues</th>
<th>The inclusive gateway throws an exception in case all conditions evaluate to false and a default flow has not been specified.</th>
</tr>
</thead>
</table>
| Workflow Patterns Support | Multi-Choice (WCP-6)  
Structured Synchronizing Merge (WCP-7)  
Acyclic Synchronizing Merge (WCP-37)  
General Synchronizing Merge (WCP-38) |
13.3.4 Event-based Gateway (Exclusive Decision (event-based))

![Diagram: 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>
<thead>
<tr>
<th>Table 13.4 – Event-Based Gateway Execution Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exception Issues</strong></td>
</tr>
<tr>
<td><strong>Workflow Patterns Support</strong></td>
</tr>
</tbody>
</table>

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

![Diagram: 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 \( x_1 + x_2 + \ldots + x_m \geq 3 \) stating that it needs 3 out of the \( m \) incoming gates to have a token in order to proceed.
prevent undesirable oscillation of activation of the Complex Gateway. ActivationCount variables should only be used in subexpressions of the form \( \text{expr} >= \text{const} \) where \( \text{expr} \) is an arithmetic Expression that uses only addition and \( \text{const} \) 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 \text{waitingForStart} = \text{true} \) and waiting for reset (\text{waitingForStart}=\text{false}).

Table 13.5 - Semantics of the Complex Gateway

| Operational Semantics | The Complex Gateway is in one of the two states: \text{waiting for start} or \text{waiting for reset}, initially it is in \text{waiting for start}. If it is \text{waiting for start}, then it waits for the activationExpression to become \text{true}. The activationExpression is not evaluated before there is at least one \text{token} on some \text{incoming Sequence Flow}. When it becomes \text{true}, a \text{token} is consumed from each \text{incoming Sequence Flow} that has a \text{token}. To determine which \text{outgoing Sequence Flow} receive a \text{token}, all conditions on the \text{outgoing Sequence Flows} are evaluated (in any order). Those and only those that evaluate to \text{true} receive a \text{token}. If no condition evaluates to \text{true}, and only then, the default \text{Sequence Flow} receives a \text{token}. If no default flow is specified an exception is thrown. The Gateway changes its state to \text{waiting for reset}. The Gateway remembers from which of the \text{incoming Sequence Flows} it consumed \text{tokens} in the first phase.

When \text{waiting for reset}, the Gateway waits for a \text{token} on each of those \text{incoming Sequence Flows} from which it has not yet received a \text{token} in the first phase unless such a \text{token} is not expected according to the join behavior of an inclusive Gateway.

More precisely, the Gateway being \text{waiting for reset}, \text{resets} when for every directed path formed by sequence flow that
- starts with a \text{Sequence Flow} \text{f} of the diagram that has a \text{token},
- ends with an \text{incoming Sequence Flow} of the Complex Gateway that has no \text{token} and has not consumed a \text{token} in the first phase, and that
- does not visit the Complex Gateway.
13.4 Events

This section describes the handling of Events.

13.4.1 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.

Table 13.5 – 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.

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.

### 13.4.2 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 Section 13.2.3.

### 13.4.3 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 Section 13.2.3.

### 13.4.4 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 retrigger 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 Section 13.2.3.

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.

13.4.5 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.

13.4.6 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

**Note** – The contents of this chapter is REQUIRED for BPMN BPEL Process Execution Conformance or for BPMN Complete Conformance. However, this chapter 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 2.

This chapter 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.
The following sections 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.

### 14.1 Basic BPMN-BPEL Mapping

This section 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

\[
\text{[ServiceTask]} = \text{Invoke Activity}
\]

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 section to denote BPMN blocks.

14.1.1 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 450 for more details of this mapping.

14.1.2 Activities

Common Activity Mappings

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

<table>
<thead>
<tr>
<th>Activity</th>
<th>Mapping to WS-BPEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>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].</td>
</tr>
</tbody>
</table>

Task Mappings

The following sections 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.

```
<invoke name="[Task-name]"
  partnerLink="[Q, Task-operation-interface]"
  portType="[Task-operation-interface]"
  operation="[Task-operation]">...
</invoke>
```

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.

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 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.

---

<KeyBasedCorrelationSetName="c-set">
  <Key name="k-name1" type="k-type1">
    <MessageKeyExpression expressionLanguage="lang1">
      expr1
    </MessageKeyExpression>
  </Key>
  ...
  <Key name="k-nameN"/>
</KeyBasedCorrelationSetName>

<invoke name="[Task-name]">
  partnerLink="[Q, Task-operation-interface]"
  portType="[Task-operation-interface]"
  operation="[Task-operation]">
    <correlations>
      <correlationSet="[Task-messageFlow-conversation-correlationKey]">
        initiate="[initialInConversation? 'join':'no']"/
      </correlationSet>
    </correlations>
  </invoke>

<prop:property name="[k-name1]" type="[k-type1]"/>
<prop:property name="[k-nameN]"/>
<prop:propertyAlias propertyName="[kName1]" messageType="[msg-name1]" part="[expr1-part]">
  <prop:query queryLanguage="[lang1]">
    expr1
  </prop:query>
</prop:propertyAlias>
...
<prop:propertyAlias propertyName="[kNameN]"/>
<correlationSets>
  <correlationSet name="[c-set]">
    properties="[k-name1] ... [k-nameN]"
  </correlationSet>
</correlationSets>
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.

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

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

```
<eventHandlers>
  <onAlarm>[timer-spec]
    <scope>[G]</scope>
  </onAlarm>
</eventHandlers>
```

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

```
<faultHandlers>
  <catch faultName="[e-fault]">
    [G]
  </catch>
</faultHandlers>
```
A **Compensation Event Sub-Process** is mapped to a WS-BPEL compensation handler as follows.

![Compensation Event Sub-Process Diagram]

### 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.

![Standard Loop Diagram]

A standard *loop* with `testTime` “After” maps as follows, where `p` denotes the *loop* condition.

![Standard Loop Diagram]

### 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).

```xml
<variable name="[counter]" type="xsd:integer"/>
...
<sequence>
  <assign>
    <copy>
      <from><literal>0</literal></from>
      <to variable="[counter]"/>
    </copy>
  </assign>
  <while>
    <condition>[p] and $[counter] &lt; [LoopMaximum]</condition>
    <sequence>
      [G]
      <assign>
        <copy>
          <from expression="$[counter]+1"/>
          <to variable="[counter]"/>
        </copy>
      </assign>
    </sequence>
  </while>
</sequence>
```

(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:

```xml
<variable name="[counter]" type="xsd:integer"/>
...
<sequence>
  <assign>
    <copy>
      <from><literal>0</literal></from>
      <to variable="[counter]"/>
    </copy>
  </assign>
  <repeatUntil>
    <sequence>
      [G]
      <assign>
        <copy>
          <from expression="$[counter]+1"/>
          <to variable="[counter]"/>
        </copy>
      </assign>
    </sequence>
    <condition>[not p] or $[counter] &gt; [LoopMaximum]</condition>
  </repeatUntil>
</sequence>
```
Multi-Instance Activities

A BPMN Multi-Instance Task with a multiInstanceFlowCondition of “All” is mapped to WS-BPEL as follows.

14.1.3 Events

Start Event Mappings

The following sections 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.

Error Start Events

An Error Start Event can only occur in Event Sub-Processes. This mapping is described on page 452.

Compensation Start Events

A Compensation Start Event can only occur in Event Sub-Processes. This mapping is described page 452.
Intermediate Event Mappings (Non-boundary)

The following sections 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.

\[
\text{[Message Intermediate Event]} \rightarrow \begin{cases} 
\text{<receive name="[e-name]" createInstance="no" partnerLink="[e-operation-interface]" portType="[e-operation-interface]" operation="[e-operation]">} \\
\text{</receive>}
\end{cases}
\]

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 of the mappings shown is chosen depending on whether the Timer Event’s TimeCycle or TimeDate attribute is used.

\[
\text{[Timer Intermediate Event]} \rightarrow \begin{cases} 
\text{<wait name="[e-name]" for="[e-TimeCycle]/"/>} \\
\text{or} \\
\text{<wait name="[e-name]" until="[e-TimeDate]/"/>}
\end{cases}
\]

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 sections 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.

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.

```
[ e ] = <throw faultName="[e-name]">
    </throw>
```

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.

```
[ e ] = <compensate> or
    <compensateScope target="[referencedActivity]">
```

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.

```
[ e ] = <exit>
    </exit>
```
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.
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.
14.1.4 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:
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.

\[
\begin{bmatrix}
\text{G1} \\
\text{G2} \\
\text{G3}
\end{bmatrix} = \text{<flow> [G1] [G2] [G3] </flow>}
\]

Sequence Pattern

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

\[
\begin{bmatrix}
\text{G1} & \text{G2} & \text{G3}
\end{bmatrix} = \text{<sequence> [G1] [G2] [G3] </sequence>}
\]

Structured Loop Patterns

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

\[
\begin{bmatrix}
\text{G}
\end{bmatrix} = \text{<while> <condition>[p] </condition> [G] </while>}
\]

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.

\[
\text{while} \quad \text{condition} = [p] \\
\text{repeatUntil} \quad \text{condition} = [\text{not } p] \\
\]

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

\[
\text{repeatUntil} \quad [G] \\
\text{condition} = [\text{not } p] \\
\]

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

```xml
<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 section, 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 section.

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

```xml
<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]"/>
        <fromPart 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**

<table>
<thead>
<tr>
<th>BPMN context access</th>
<th>BPEL context access</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getDataobject(dataObjectName)</code></td>
<td><code>[$dataObjectName]</code></td>
</tr>
<tr>
<td><code>getProcessProperty(propertyName)</code></td>
<td><code>{{$processName}.propertyName} where the right process-Name is statistically derived.</code></td>
</tr>
<tr>
<td><code>getActivityProperty(activityName, propertyName)</code></td>
<td><code>[$activityName.propertyName]</code></td>
</tr>
<tr>
<td><code>getEventProperty(eventName, propertyName)</code></td>
<td><code>[$eventName.propertyName]</code></td>
</tr>
</tbody>
</table>
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.

14.2 Extended BPMN-BPEL Mapping

Additional sound BPMN Process models whose block hierarchy contains blocks that have not been addressed in the previous section 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 Section 15.1 SHOULD be used where applicable.

14.2.1 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

14.2.2 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

14.2.3 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”
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 section 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”
14.2.4 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

14.2.5 BPMN Elements that Span Multiple WSBPEL Sub-Elements

Figure 14.8 below 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](image-url)
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 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.cmf.
- 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"?>

    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: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

```xml
<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:

```xml
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 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 specification.

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 specification (in Chapter 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.
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
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
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**

  
  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 defined 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 its 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 (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
- \(? + /\text{ownedLabel} : \text{Label} \[*\] \{\text{readOnly, union, subsets \text{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 get 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 and 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 (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.
<table>
<thead>
<tr>
<th><strong>Collapsed Sub-Process</strong></th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compensation Flow</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Compound Activity</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Controlled Flow</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Decision</strong></td>
<td>A gateway within a business process where the Sequence Flow can take one of several alternative paths. Also known as &quot;Or-Split.&quot;</td>
</tr>
<tr>
<td><strong>End Event</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Event Context</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Exception</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Exception Flow</strong></td>
<td>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).</td>
</tr>
<tr>
<td><strong>Expanded Sub-Process</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
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 known 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.
**Normal Flow**
A flow that originates from a Start Event and continues through activities on alternative and parallel paths until reaching an End Event.

**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.

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

**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.”

**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.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Transaction</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Trigger</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Uncontrolled Flow</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
INDEX

A
Abbreviations 17
Abstract Task 430, 449
Activities 426
Activity 151
Activity Boundary Connections 258
Activity Service 13
Ad-Hoc Sub-Process 180, 431
Artifacts 66
Attributes for Boundary Events 257
Auditing element 311

B
BPEL Process Execution Conformance 1, 10
BPEL Process Execution Semantics Conformance type 1
BPEL4People 13
BPMN 1.2 22
BPMN Complete Conformance 1
BPMN Extended Modeling Elements 31
BPMN Process Modeling Conformance 8
BPMNDiagram 370
BPMNEdge 375
BPMNLabel 376
BPMNLabelStyle 377
BPMNPlane 371
BPMNShape 372
Business Process Definition Metamodel 13
Business Process Management (BPM) 21
Business Process Modeling 13
Business Rule Task 163, 430
Business Transaction Protocol 13

C
Call Activity 183
Call Activity Mapping 218
Call Choreography 333
Call Conversation 131
CallableElement 186
Cancel Events 263
Cancellation 235
CatchEvent 236
Changes from BPMN, v1.2 479
Choreographies 25
Choreography 315
Choreography Activity 322
Choreography Conformance 10
Choreography Conformance type 1
Choreography Elements 10
Choreography Modeling Conformance 1
Choreography Semantics 11
Choreography Task 323
Choreography Types 10

Collaboration 109
Collaboration element 2
Compensation 235, 302, 441
Compensation Events 263
Compensation handler 303, 441
Compensation triggering 442
CompensationEventDefinition 264
Completing 429
Complex Gateway 295, 361, 437
ComplexBehaviorDefinition 194
Compliance 1
Conditional Sequence Flows 320
ConditionalEventDefinition 265
Conformance 1
Context-based correlation 78
Conventions 16
Conversation 124, 426
ConversationLink 134
ConversationAssociation 135
ConversationNode 128
Conversations 26
Core Elements 49
Core Structure 49
Correlation 74, 136
CorrelationKey 76

D
Data Input 213
Data Object 205
Data Object Reference 206
Data Output 215
DataAssociation 221
DataInputAssociation 224
DataOutputAssociation 224
DataState 206
DataStore 208
DD architecture 481
Default Sequence Flows 320
Descriptive conformance sub-class elements 3
Diagram Common (DC) package 483
Diagram Commons (DC) 367
Diagram Definition (DD) 367
Diagram extensions 8
Diagram Interchange (DI) 367
Diagram Interchange (DI) package 487
Diagram Point of View 26
Diagram Types 41
Documentation element 56
Dublin Core Meta Data 13

E
ebXML BPSS 13
Elements 27
End Event 246
End Events 343, 443
EndPoint element 105
R
RDF 14
Receive Task 161, 430, 448
Receive Task Mapping 217
References 12
Relationship element 62
Resource assignment 154
Resource class 95
ResourceAssignmentExpression element 155
ResourceParameterBinding element 155
ResourceRole element 154
Resources 154
RFC-2119 12
RootElement 64

S
Scope 1, 21, 49, 109, 124, 145, 315, 367, 425, 445, 475
Scopes 281
Script Task 164, 430
Script Task Mapping 218
Send Task 159, 430, 449
Send Task Mapping 217
Sequence Flow 97
Sequence Flow Connections 153, 259
Sequence Flows 427, 461
Service package 104
Service Task 158, 430, 448
Service Task Mapping 217
shapes 8
SOAP 1.2 14
StandardLoopCharacteristics 190
Start Event 238, 426, 439
Start Events 339
Structural Conformance 8
Sub-Choreography 328
Sub-Conversation 130
Sub-Process 430
Swimlanes 363

T
Task 156, 430
Task Mappings 448
Terminate Event 273
Text Annotation 71
ThrowEvent 237
Timer Event 274
Token 427
Transaction 178
trigger 235

U
UDDI 14
URI 14
User Task 163, 430
User Task Mapping 217

V
Visual Appearance 11
Visual Interchange 11
visual language 8

W
Web Services Business Process Execution Language (WSBPEL) 1
Web Services Transaction 14
WfMC Glossary 14
Workflow Patterns 14
WS-BPEL 445
WSBPEL 15, 17
WS-Coordination 15
WSDL 15, 17
WS-HumanTask 15

X
XMI 477
XML 1.0 (Second Edition) 15
XML-Namespaces 15
XML-Schema 15
XPath 15
XPath Expressions 226
XPDL 16
XSD 475