Institute of Architecture of Application Systems (IAAS)
University of Stuttgart

Kerstin Pfitzner

Diploma Thesis

Nr. 2618

"Choreography Configuration for BPMN"

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Frank Leymann
Supervisor:	Dipl.-Inf. Oliver Kopp M. Sc. Gero Decker
Commenced:	June 1st, 2007
Completed:	December 1st, 2007
CR-Classification:	D 2.3, C.2.4, H.4.1

Contents

1 Introduction to Process Choreographies 7

1.1 BPMN 8

1.2 BPEL4Chor 10

2 Adapting BPMN for Choreographies (BPMN+) 12

2.1 Modeling BPEL4Chor with BPMN 12

2.2 Meta-model for BPMN+ 17

2.3 Element attributes 27

2.4 BPMN+ Example 49

3 BPMN+ to XPDL4Chor 52

3.1 XPDL Analysis 52

3.2 Mapping BPMN+ Elements to XPDL4Chor 54

4 BPMN+ to BPEL4Chor 59

4.1 BPMN to BPEL 59

4.2 Generating the BPEL4Chor Processes 60

4.3 Generating the BPEL4Chor Topology 77

4.4 Implementation of the Transformation 84

5 Editor Extension 86

5.1 Oryx Editor 86

5.2 BPMN+ Stencil Set 87

5.3 Calling the Transformation Web Service 93

6 Summary and Outlook 99

7 Bibliography 100

A BPMN+ Element Attributes 104

A.1 Business Process Diagram 104

A.2 Process 105

A.3 Graphical Object 106

A.4 Swimlane 106

A.5 Artifact 108

A.6 Connecting Object 111

A.7 Flow Object 112
A.8 Activity 117
A.9 Supporting Types 122

Erklärung 129

List of Figures

Figure 1.1: Example of a collaboration business process in BPMN.................................. 9
Figure 2.1: (a) if, (b) pick and (c) flow with parallel and exclusive gateways................. 14
Figure 2.2: A pool set, a pool and a sub-process.. 19
Figure 2.3: Events... 20
Figure 2.4: Task activities.. 20
Figure 2.5: Gateway types... 21
Figure 2.6: (a) fault handler and (b) compensation handler for synchronous service activity and (c) event handlers in scope.. 22
Figure 2.7: Termination handler within a scope or a parallel multi-instance loop........... 22
Figure 2.8: Standard, message, fault and counter variable data objects........................... 23
Figure 2.9: (a) participant reference and (b) participant set data object........................... 24
Figure 2.10: Associations to communicating activities.. 25
Figure 2.11: Associations from communicating activities.. 25
Figure 2.12: Participant data objects and multi-instance loops.. 26
Figure 2.13: Containment associations for participant data objects................................ 26
Figure 2.14: Participant reference passed over a message flow....................................... 27
Figure 2.15: Requesting the price of a book in BPMN+.. 51
Figure 4.1: Non-safe sequence flow.. 62
Figure 4.2: Invalid combination of multiple start evens.. 65
Figure 4.3: Mapping inclusive decision gateway onto Petri net modules [WEvdA+05].. 75
Figure 4.4: Generating the type of participant references from associations.................... 80
Figure 4.5: Generating the type of participant sets from associations............................. 81
Figure 4.6: Implicit containment... 82
Figure 4.7: Message flows with the same source generated to one message link............. 83
Figure 4.8: Message flows with the same target generated to one message link.............. 83
Figure 4.9: Main packages of the transformation web service.. 84
Figure 5.1: User interface of the Oryx editor.. 87
Figure 5.2: Complex property dialog for Correlations... 91
Figure 5.3: Transformation plugin in the toolbar... 94
Figure 5.4: Result dialog for the transformation... 97
Figure 5.5: Extensions in the Oryx architecture.. 98

List of Listings

Listing 3.1: BPMN+ properties with basic and XML schema type in XPDL.................. 53
Listing 3.2: BPMN+ property for correlation in XPDL4Chor... 55
Listing 3.3: A pool set in XPDL4Chor... 55
Listing 3.4: Variable data object in XPDL4chor.. 56
Listing 3.5: Event-based decision and merge gateway in XPDL4Chor............................ 57
Listing 3.6: Empty task in XPDL4Chor.. 57
Listing 4.1: Example for generated participant types... 79
Listing 5.1: SVG file for a BPMN+ pool set 88
Listing 5.2: “Layout” function for a pool set.. 89
Listing 5.3: “Serialize” function for a BPMN+ stencil.. 90
Listing 5.4: Example definition of a complex property... 92
Listing 5.5: Offering the plugin functionality.. 94
Listing 5.6: Excerpt from serialized pool set properties.. 94
Listing 5.7: XSLT template for generating the XPDL4Chor selects element.................. 95
Listing 5.8: XSLT processing with JavaScript... 96
Listing 5.9: SOAP call with JavaScript.. 96

1 Introduction to Process Choreographies

Service-oriented Architecture (SOA) is an architectural style for heterogeneous distributed systems based on services. Services are loosely coupled components and can be orchestrated for implementing business processes. In the web service platform architecture ([CLS+05]) the Business Process Execution Language (BPEL, [BPEL07]) is used as standard for describing *business processes orchestrations*.

Business processes are more and more running over a long period of time. This results in business processes that are engaged in long running conversations with other processes. To handle these complex conversations, a new view point on interactions between business processes is needed. This view point describes the interactions from a global point of view and not from a process perspective like in orchestrations. The resulting global models are called *process choreographies*.

As stated in [Wesk07], a process choreography is used for expressing a business-to-business collaboration. It provides a model that specifies the nature of a collaboration. In this way it defines an agreement on how business partners need to interact with each other. Thus, a choreography is the starting point for developing the executable process orchestrations for each business partner.

There can be distinguished two different modeling approaches for choreographies: interaction models and interconnected interface behavior models. *Interaction models* are built up of elementary interactions. Dependencies between these interactions are grouped into more complex interactions. The Web Service Choreography Description Language (WS-CDL, [KBR+05]) and Let's Dance ([ZBDtH06]) are representatives for defining choreographies following this approach. *Interconnected interface behavior models* define the control flow for each participant that is taking part in the choreography. Interactions between these participants are defined based on this control flow. BPEL4Chor ([DKLW07]) and BPMN ([BPMN06]) can be used for defining choreographies this way. BPMN also allows the modeling of interaction models, even though it is not common usage.

Several roles are involved in the development of a process choreography: business engineers, system architects and developers. Business engineers mainly take part in the choreography design phases. System architects bridge the design and implementation phases. Finally, developers are responsible for realizing the process orchestrations based on the designed choreography. While business engineers think visual, developers mostly work with textual representations. A graphical notation, which illustrates the elements of a textual language, could be used during the development life cycle to enhance the communication between the involved roles.

Like BPEL, BPEL4Chor lacks a graphical representation, whereas BPMN is a typical visual notation for business processes. Since BPEL4Chor and BPMN support the same modeling approach, this thesis integrates both to enable a visual modeling of BPEL4Chor choreographies. BPMN does not provide the definition of detailed technical configurations like BPEL4Chor. However, this thesis targets the generation of fully defined BPEL4Chor choreographies where no further refinement is necessary. That is why a choreography configuration for BPMN is introduced. Partial results of this thesis have been published in [PDKL07].

In the next chapter a short survey of the used languages BPMN and BPEL4Chor is given. After that, BPMN is investigated concerning the modeling of BPEL4Chor choreographies and a configuration of BPMN for this purpose is introduced in chapter 2. Chapter 2.4 and 4 describe the transformation of the configured BPMN diagrams to BPEL4Chor. To integrate the configured BPMN and the transformation, an existing BPMN editor is adapted with them in chapter 5.

1.1 BPMN

The Business Process Modeling Notation (BPMN, [BPMN06]) was developed by the Business Process Management Initiative (BPMI) that released version 1.0 in May 2004. It is now maintained by the Object Management Group (OMG). A BPMN specification version 2.0 is currently proposed ([BPMN07]). The primary goal of the notation is to be understandable by all business users.

BPMN allows the creation of process segments as well as end-to-end business processes. There can be distinguished three types of sub-models in a BPMN model:

- Private (internal) business process: internal to a specific organization
- Abstract (public) process: represents interactions between a private business process and another process or participant
- Collaboration (global) process: depicts the interactions between two or more business entities

Figure 1.1 illustrates an example of a collaboration process that represents the following use case: A client requests the price of a book from a bookshop. The bookshop requests the price from each supplier that he knows to offer this book. Receiving this request causes a supplier to calculate the price for this book. If the supplier has this book in his assortment, he sends the price back to the bookshop. If the supplier does not know this book, he sends a fault message back to the bookshop. Receiving a fault message, the bookshop starts a fault handling, where he updates his supplier data. In this way he will not request the supplier for the price of this book again. After all suppliers have replied either a fault or the price, the bookshop verifies if any supplier has responded a price. If so, the bookshop determines the cheapest supplier and sends the according price to the client. Otherwise, the supplier responds an error message to the client. Meanwhile, the client waits for the response of the bookshop. If the client receives a response, the use case ends. The various diagram elements of this model are explained in the following.

The top-level element of a BPMN model is a *Business Process Diagram* (BPD). It is made up of a set of other graphical elements that build the model. These elements are organized into specific categories so that the reader of a BPD can easily recognize the basic element types. Certain variations and information can be added to these elements without changing the basic element shapes. There are four element categories: flow objects, connecting objects, swimlanes and artifacts.

Flow objects define the behavior of a process in a BPD. There are three flow object types: Events, gateways and activities. An *event* is visualized as circle and is something that “happens” during the process. They usually have a cause (trigger) or an impact (result) that affects the flow of the process. Events can be located at the start of the process (start event), in between the process activities (intermediate event) or at the end of the process (end event). *Gateways* are visualized by a diamond shape. They are used to determine decisions, forking, merging and joining of paths within a process depending on the gateway type (exclusive, parallel, inclusive, complex). *Activities* are drawn as rounded rectangles and represent work that is performed within a process. The types of activi-

ties are: tasks (atomic) and sub-processes (compound). Activities are repeated multiple times if they have a standard or multi-instance loop marker (the bookshop in Figure 1.1 contains a multi-instance loop that is marked with a pair of vertical lines in parallel).

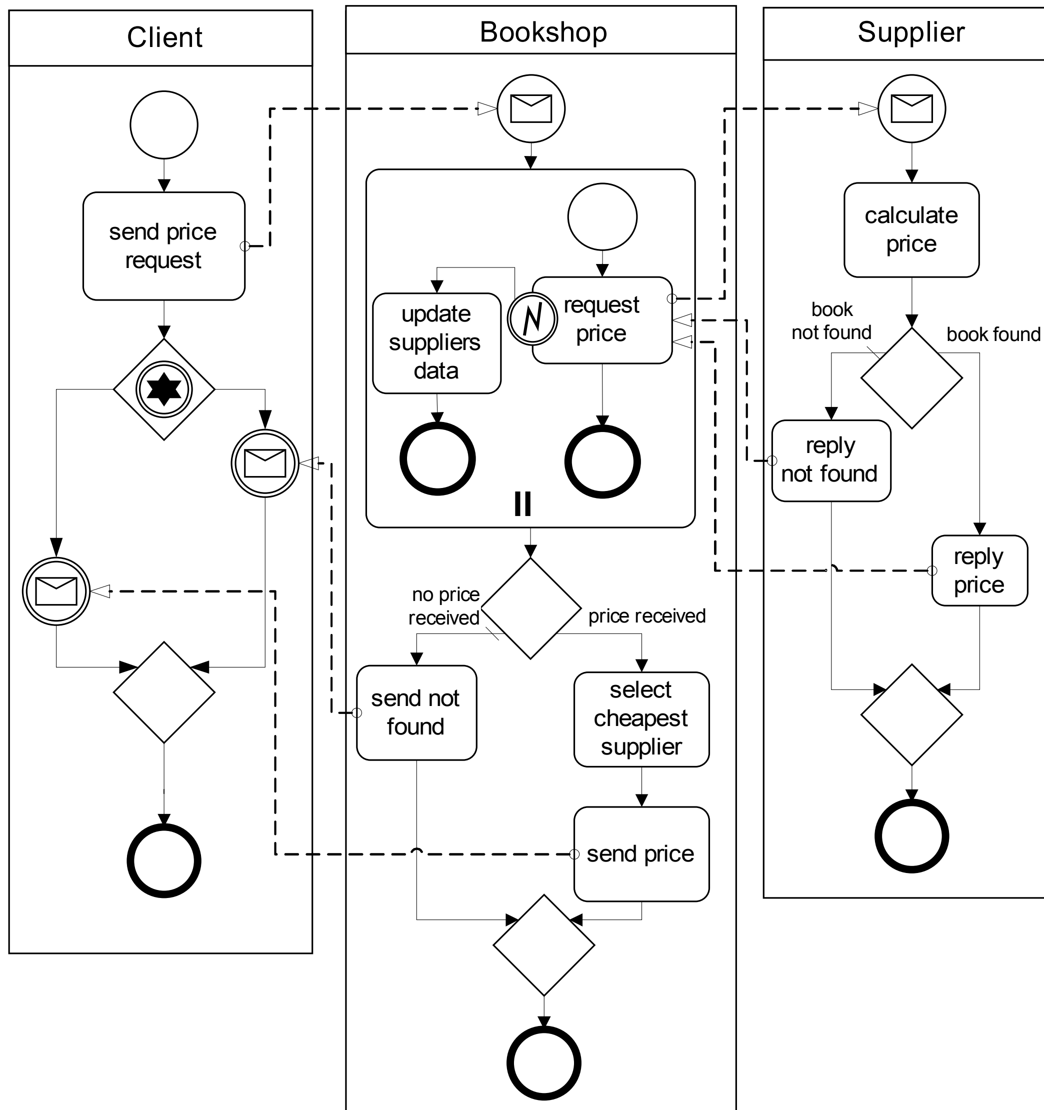


Figure 1.1: Example of a collaboration business process in BPMN

Artifacts show additional information in the diagram that is not directly related to the sequence flow or message flow of the process. BPMN provides standard artifacts called data object, group and annotation. In addition to this, other artifact types may be added by the modeler or modeling tool vendors. *Data objects* show what data is required or produced by activities. The direction of the connected association indicates whether the data is used as input or output. A *group* is represented by a rounded rectangle with a dashed line. It groups activities for documentation or analysis purposes. *Annotations* provide additional text information for the reader of the BPD. It is associated with diagram elements using non-directed associations.

Flow objects and artifacts are connected together using c*onnecting objects.* There are different types of connectors that connect different types of flow objects and artifacts. A *sequence flow* is represented by a solid line with a solid arrow head. It determines the order that activities are performed in a process. A *message flow* is represented by a dashed line

with an open arrow head. It illustrates the communication between separate participants. This means that it shows the sending and receiving of a message by connecting the message flow with a sending and a receiving activity. An *association* is represented by a dotted line and associates artifacts with flow objects. Associations can be directed (arrowhead is present) or non-directed (no arrowhead).

Swimlanes are used to partition and organize the activities of a BPD. There are two types of swimlanes in BPMN: pools and lanes. A *pool* is a graphical container for a set of lanes and represents a participant in a BPD (e.g. the client in Figure 1.1). Thus, it is used when the diagram involves two or more different participants. The activities within a pool build up a process and the sequence flow between these activities may not cross the pool boundary. A *lane* is a sub-partition within a pool and acts as graphical container for a set of activities. The activities can be organized and categorized within a pool using multiple lanes. Both, pools and lanes, are visualized as rectangular boxes.

A goal of BPMN is to bridge the gap from the business-oriented process modeling notation to process execution languages. For this purpose the graphical elements can specify special attributes that convey the information required to execute a business process. In addition to this, there is described an informal mapping of BPMN to BPEL using these attributes.

1.2 BPEL4Chor

BPEL4Chor is an extension of BPEL for expressing choreographies and was first introduced in [DKLW07]. It is based on the *Abstract Process Profile for Observable Behavior* described in [BPEL07] and adds an interconnection layer on top of this profile. A BPEL4Chor choreography is built up of three components that are related with each other:

- Participant behavior descriptions: define the control flow dependencies between the activities of the participants that are taking part in the choreography
- Participant topology: defines the participant types, participant references, participant sets and message links that build the structure of the choreography
- Participant grounding: defines the technical configuration of the choreography

A *participant behavior description* (PBD) is an abstract BPEL process that can hide implementation details. For this purpose, a PBD can contain opaque activities in addition to the normal BPEL activities. A PBD is decoupled from the technical configuration. That is why the attributes `partnerLink`, `portType` and `operation` are omitted in communicating activity elements. A `forEach` does not need to specify a counter. In this case the loop is iterating over a set of participants. The association between the participant set and the `forEach` has to be defined in the participant topology.

The *participant topology* lists participant types. The behavior of a participant type is determined by an appropriate PBD. *Participant references* are defined in the topology, too. They have to be associated with one of the defined participant types. A participant reference is bound to a participant at runtime. If there are multiple participants of the same type involved, they can be represented as *participant set* – especially if the exact number of participants is not known at design time. *Message links* are also defined in the participant topology. They connect the communicating activities of different participant types and express a message exchange between the activities. To identify the activities connected with the message links, communicating activities must have an identifier that is unique within the PBD. As the knowledge about participants is local, *link passing mobility*

is established for message links. In this way participant references can be passed over a message link to another participant.

The technical configuration is omitted in the PBD and the participant topology. In this way it is decoupled from the actual choreography. The mapping to web-service specific configurations is done in the *participant grounding*. It defines the port type and operation for each message link. The location of passed participant references in a message is specified using existing property aliases. In addition to this, correlation properties are grounded to concrete WSDL properties. The grounding can later be used for the executable completion of the PBDs to generate executable BPEL processes.

2 Adapting BPMN for Choreographies (BPMN$^+$)

In the previous chapter the modeling notation BPMN and the XML based choreography language BPEL4Chor were introduced. Like BPEL, BPEL4Chor is not associated with a graphical representation, whereas the collaboration processes of BPMN provide the ability to visualize choreographies. Thus, the lack of a graphical representation for BPEL4Chor might be effaced with BPMN. To clarify this, the applicability of BPMN for modeling choreographies has to be investigated. This investigation is described in section 2.1. After that, necessary extensions for the BPMN meta-model elements and attributes will be introduced in section 2.2 and 2.3. These extensions enable the modeling of all BPEL4Chor constructs.

2.1 Modeling BPEL4Chor with BPMN

Designing a BPEL4Chor choreography ([DKLW07]) generally takes place in several steps. In top-down development approaches the main aspects of the global model are the starting point and the interactions between the participants are developed. Details of the participant behavior are hidden using opaque activities. After that, the participant behavior descriptions get refined. The opaque activities are replaced with the actual functionality, variable definitions are added and the error handling is described. A graphical representation for BPEL4Chor would ease the development in all these steps, since it is comprehensible for all business users. That is why a graphical representation for all BPEL4Chor elements is needed.

In this section BPEL4Chor is inspected for elements that must be provided by a graphical notation like BPMN. In doing so, it will be examined how these elements can be represented in BPMN and in which way BPMN has to be extended for this purpose.

In [DePu07] the weaknesses of BPMN for choreography modeling were already investigated based on the Service Interaction Patterns [BDtH05]. Also extensions were proposed to overcome these limitations. BPEL4Chor supports most of these patterns (see [DKLW07] for more details). Thus, the deficiencies of BPMN and the proposed extensions described in [DePu07] will appear also in this investigation. In addition to this, BPEL4Chor specific weaknesses will be detected and appropriate configurations in BPMN will be proposed.

BPEL4Chor uses abstract BPEL processes to model the behavior of the choreography participants. So first of all, modeling BPEL4Chor means to model abstract BPEL processes. As stated in [ReMe06] there is a conceptual mismatch between BPMN and BPEL, because they are employed in different BPM life cycle stages. While BPMN is used in the early stages, when the process gets developed, BPEL is more concentrated on the execution of a business process. That is why there are some constructs missing in BPMN regarding the execution of a business process.

The basic construct of BPEL4Chor processes are the different activity types. Those activities are mainly organized in a block-structure where the sequence is determined by the nesting of the activities using structured activities. Moreover, the `flow` activity allows to arrange activities in a graph structure connecting them using control links. The behavioral context of activities can be defined using scopes. A `scope` influences the execution behavior of its enclosed activities by defining variables, handlers, message exchanges and correlation sets.

In contrast to BPEL4Chor, BPMN has mainly a graph structure where the order of the activities is determined by a sequence flow. BPMN provides different flow objects that can be connected with sequence flow. They can be used for modeling the BPEL4Chor activities:

A task is an atomic activity included within a process. As can be seen in the table below, BPMN already provides some task types that can be reused for modeling the basic activities. However, the basic activities `assign`, `validate` and `empty` do not have an equivalent task type in BPMN.

BPMN Task	BPEL4Chor Activity
service task	synchronous `invoke`
receive task	`receive`
send task	`reply` asynchronous `invoke`
non-typed task	`opaqueActivity`
-	`assign`
-	`validate`
-	`empty`

Events are something that "happens" during the flow of the process in BPMN. They can be located at the start, within or at the end of the process flow. There are multiple alternatives, how these events can be triggered or trigger something themselves. Because in BPEL4Chor it can be left open what triggers the instantiation of a process, a non-triggered start event can be used to start a process. If the process is instantiated by the receipt of a message, this can be expressed with a start event triggered by a message. In addition to this, the table below shows the BPMN events that can be used to express BPEL4Chor activities. The end of a process does not cause any events in BPEL4Chor. Therefore, the non-result end event is used to indicate the end of a process. Other triggers (e.g. rule, link) must not be used for modeling BPEL4Chor with BPMN, because they have a semantic that is not part of BPEL4Chor.

BPMN Event	BPEL4Chor Activity
message start event	`receive` (createInstance="true") `onEvent`
timer start event	`onAlarm`
message intermediate event	`receive` (createInstance="false")
timer intermediate event	`wait`
error intermediate event	`throw` or `rethrow`
compensation intermediate event	`compensate` or `compensateScope`

Because of the graph structure BPMN is mainly based on, structured activities of abstract processes have to be expressed using the sequence flow. In [ODtHvdA07] it is already shown how this can be done using gateways. A `sequence` is just the line-up of flow objects connected with at most one incoming and one outgoing sequence flow and without using gateways. For modeling the `if` activity a data-based exclusive gateway is used to

illustrate the different branches and the exclusive merge is used to merge the branches again. The pick activity can be modeled using an event-based exclusive decision gateway. The `onMessage` branch is represented by an intermediate message event or a receive task. The `onAlarm` branch is represented by an intermediate timer event. Pick branches are merged using an exclusive merge. Concurrent branches of a `flow` activity are created using a parallel fork gateway and synchronized with a parallel join gateway. The loops `repeatUntil` and `while` can be modeled with exclusive gateways in a structured arrangement. Figure 2.1 Illustrates a BPEL4Chor `if`, `pick` and `flow` in BPMN. The graph structure represented by a `flow` activity has to be located between a splitting and a merging gateway, too. However, in this case data-based exclusive, inclusive or parallel split gateways and exclusive, inclusive or parallel merge gateways can be used. The different options for modeling a `flow` activity are explained in more detail in section 4.2.2.

Despite the graph structure, BPMN provides the nesting of activities using sub-processes. An embedded sub-process is a compound activity that contains other activities. As stated in [BPMN06] an embedded sub-process maps to a `scope` activity. Moreover embedded sub-processes can illustrate multi-instance or standard loops. This is applicable for the `while`, `repeatUntil` and `forEach` activities. A `forEach` can be parallel, thus it is illustrated using a multi-instance loop, whereas `while` and `repeatUntil` are illustrated as standard loops. In BPEL4Chor scopes can be isolated, which means that they provide control of concurrent access to shared resources. As stated in [RtHvdAM06] this can be used for modeling workflow patterns like “interleaved routing” and “critical section” that are supported by BPEL4Chor. BPMN does not provide the semantics of isolated scopes for sub-processes, so these patterns are not supported in BPMN.

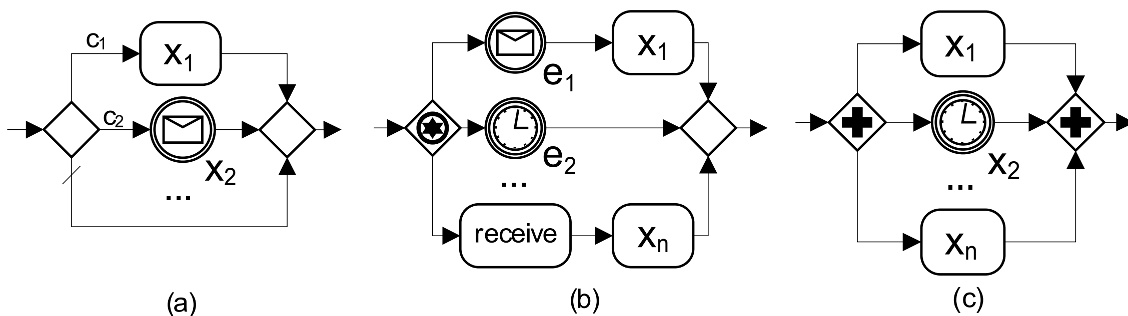


Figure 2.1: (a) `if`, (b) `pick` and (c) `flow` with parallel and exclusive gateways

The `extensionActivity` has to be modeled as intended by the extension activity designer and depends on the definition of the extension. It is not possible to foresee all future extension activities. Thus, this thesis is focused on BPEL4Chor without extension activities. However, it is allowed to plug in extension activities if they provide a graphical representation, which conforms to the BPMN specification, and a mapping to BPEL4Chor.

In addition to the basic activities, there can be event handlers defined in BPEL4Chor that run concurrently to the normal flow of the activities within a scope or process. Those event handlers can be triggered by messages or alarms. Furthermore, fault handlers can be defined for a `scope`, an `invoke` activity or a `process`. A compensation handler can be defined for a `scope` or an `invoke` activity.

In BPMN there is already a mechanism for defining fault and compensation handlers using compensation and error intermediate events attached to the boundaries of a sub-process or task. The actual fault and compensation handling is done by the flow objects

that follow the attached event. A triggered attached event creates an event context, which interrupts the activity, the event is attached to and the flow is redirected through the attached event. That is why the attached events are not applicable for event handlers in BPEL4Chor, because those run concurrently to the process or scope they are defined in. Sub-processes, as defined in the BPMN specification, are not applicable for event handlers, too. If a sub-process has an incoming sequence flow, its instantiation is predetermined by its position in the process flow. A sub-process with no incoming sequence flow is only allowed if the parent process does not define a start event. In this case it is instantiated when the parent process is instantiated. Thus, BPMN sub-processes can not be instantiated by the receipt of a message or by an alarm. So there is no way to define event handlers in BPMN with such a behavior so far.

The BPMN specification does not clearly state what is happening during the interruption of an activity when the event context responses to a trigger. In BPEL4Chor all activities nested in this activity will be terminated if a fault event occurs. This termination can trigger a handler again. Such a termination handler can also be used in parallel `forEach` activities that have a completion condition defined. BPMN does not provide a mechanism to express the triggering of an activity termination. It only provides a cancel trigger that can exclusively be used for transactional sub-processes. Transactional sub-processes have a special behavior that is controlled through a transaction protocol (e.g. BTP[1] or WS-Transaction[2]). When using BPEL4Chor, transaction protocols are defined as part of the exchanged messages and not in a `scope`. However, the message definitions are not part of BPEL4Chor itself, because they are part of the related WSDL files. That is why transactional sub-processes must not be used for modeling choreographies.

In BPEL4Chor the data, which is processed by the activities, is stored in variables. To express these elements, BPMN provides data objects. They can be associated with flow objects, whereas the direction of the association indicates the data object as input or output variable. BPEL4Chor knows different variable types that differ in their type declaration (e.g. counter variable in a loop, fault variable of a fault handler). However, BPMN does not distinguish different types of data objects.

For defining the static structure of the choreography, BPEL4Chor introduces a participant topology that describes the different participants and the communication between them. Because there can be more than one participant with the same behavior (e.g. different bidders taking part in an auction), a participant type is introduced. Participant types can be associated with participant references, which are bound to concrete participants during runtime. The abstract processes mentioned above describe the behavior of the participant types. Participant sets are introduced for scenarios, where multiple participants of the same type may take part in a conversation, especially when the actual number of participants is not known at design time.

BPMN already knows a participant as business role and illustrates it in the form of a pool, which is a container for a process. In the case of BPEL4Chor, a pool represents the participant type and is the container for the abstract process that describes the behavior of the type. As already stated in [DePu07] there is no multiplicity in the notion of a pool. Each pool is a participant in the participant topology of the choreography. Participant multiplicity, which is needed for expressing participant sets, is not supported in BPMN. This leads to the fact that the whole participant hierarchy, including participant sets, cannot be expressed. Moreover, the participants in BPEL4Chor are just references to actual participants, whereas BPMN does not know references to participants.

1 See http://www.oasis-open.org/committees/business-transactions
2 See http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html

Communication between the participants is described using message links in BPEL4Chor. They specify which participant can potentially send a certain message to which other participant. Because the information about participants is local, participant information can be passed over a message link (link passing mobility).

In BPMN, message flows can be used to express the message links. They illustrate the communication between the activities of different pools. The sending and receiving participant of a message flow can be expressed using data objects. They are associated with the communicating activities to distinguish the sender and the receiver of a message as described in [DePu07]. If the data objects are associated with a message flow, this expresses that a participant is passed over the message flow. The participant data objects have to be distinguished from the data objects that represent variables.

The communicating service tasks, send tasks and receive tasks can be distinguished from each by the connected message flows. A service task has at least one outgoing and at least one incoming message flow, a send task has no incoming message flow and a receive task has no outgoing message flow. Nevertheless, other task types cannot be distinguished from each other using the normal BPMN task illustration.

In BPEL4Chor the participant behavior is free of technical configuration details to achieve a higher reusability of the choreography models. A special grounding file adds web-service-specific details to the choreography.

BPMN already provides a set of attributes that can be mapped to BPEL as described in [Whi05] and [BPMN06]. But these attributes are not sufficient for generating the complete BPEL4Chor information. For example, `import` elements of a process or the `from` and `to` elements in a variable declaration can be neither expressed in a graphical way nor as attributes of BPMN elements.

BPMN provides attributes for the flow objects that hold the implementation details. Since in BPEL4Chor these details should be separated from the actual choreography the grounding cannot be expressed in the BPMN way.

It can be concluded that the basic ideas of process choreographies are already implemented in BPMN. But having a closer look, there are some deficiencies:

- basic tasks assign, validate and empty are not present
- semantic of isolated scopes is missing
- no distinction between different data object types
- multiplicity of participants cannot be expressed
- event handlers triggered by messages or alarms are not supported
- termination handler cannot be illustrated
- lack of attributes
- grounding cannot be decoupled from the choreography

Because of these weaknesses, additional elements and attributes have to be introduced for BPMN to enable the modeling of BPEL4Chor choreographies. To conform to the BPMN specification, the following possibilities for extensions can be summarized:

There can be added new markers associated with the BPMN elements. Moreover, it is allowed to change the line style of existing elements. However, it has to be ensured that the changed shape will not conflict with any current shape and that the basic shapes will remain unchanged. There is not stated whether and how attributes of BPMN elements can

be changed. So it is assumed that additional attributes can be defined for every element and existing attributes can be restricted.

In addition to the elements described above, BPMN provides a lot more constructs. These constructs are not present in BPEL4Chor or can be expressed using other BPMN constructs (e.g. complex gateways). That is why the following elements must not be used for modeling BPEL4Chor choreographies:

- complex gateways
- ad-hoc and transactional sub-processes
- start events with the trigger link, rule and multiple
- all end events except the non-triggered
- cancel, rule, link, multiple or non-triggered intermediate events
- user, script, manual or reference tasks

Groups, text annotations and lanes are elements that support the graphical representation of the diagram. They can be used by the modeler to enhance the comprehensibility for the human reader.

2.2 Meta-model for BPMN$^+$

The previous section has shown which elements are missing in BPMN and which element needs to be adapted to model a BPEL4Chor choreography with BPMN. Moreover, the possibilities were described that BPMN provides to adapt existing elements and to introduce new ones. Based on these statements, the following section introduces an extension of BPMN for modeling choreographies in the terms of BPEL4Chor. The notion BPMN4chor was already introduced in [Deck06]. BPMN4Chor focuses on role diagrams and interaction diagrams, whereas the extension presented in this thesis deals with interconnected interface behavior models. Therefore, this section introduces the term BPMN$^+$. In the following, the complete set of elements and their graphical representation is described that can be used in BPMN$^+$. This includes existing BPMN elements, adapted BPMN elements and elements that are introduced in BPMN$^+$. The attributes of these elements can be found in section 2.3.

2.2.1 Business Process Diagram

The business process diagram, in short BPD, comprises the BPMN$^+$ diagram and the general attributes for the choreography. It has no graphical representation, because it acts as container for the other BPMN$^+$ elements. So all elements defined for the choreography belong to a BPD. The attributes of a BPD can be found in section 2.3.1.

2.2.2 Swimlane

Swimlane is a generic term for pools, pool sets and lanes. Pools and pool sets are used to define the participant types of the participant topology. A pool set is introduced in BPMN$^+$ and inherits all characteristics of a pool, which is already part of BPMN. Each pool and pool set is a container for a process, which describes the participant behavior description of the participant type. In contrast to the pool, which represents a single participant reference in the participant topology, a pool set indicates that there are multiple participants of the same type. The actual participants have to be expressed using participant reference and participant set data objects (see section 2.2.5).

The graphical representation of a BPMN pool does not change in BPMN$^+$. Figure 2.2 illustrates that a pool set is drawn as a shaded pool. The attributes of a pool and pool set can be found in section 2.3.4 and 2.3.5. Lanes can be used in BPMN$^+$ as well as in BPMN, but only for layout purposes.

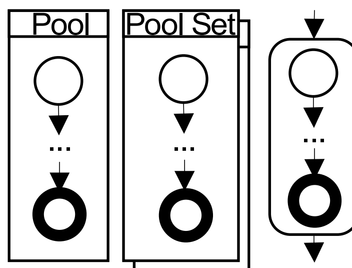


Figure 2.2: A pool set, a pool and a sub-process

2.2.3 Process

A process holds flow objects and the control flow between them. It expresses the activities performed by a choreography participant of a certain type. In the diagram a process is embedded either in a pool or a pool set and may contain sub-processes. Each process can have multiple start events and multiple end events. A process can be instantiated by the receipt of a message or the instantiation of a process is not defined. That is why only message triggered or non-triggered start events are allowed in a process. The sequence flow connecting the flow objects of a process is not allowed to cross the boundaries of the pool or pool set that the process belongs to. The attributes of a process can be found in section 2.3.2.

2.2.4 Flow Object

A flow object can be an activity, an event or a gateway. An activity again can be a task or a sub-process. The attributes of flow objects can be found in section 2.3.15.

Event

As provided in BPMN, events are also part of BPMN$^+$. A message start event expresses the receipt of a message that instantiates the process. It can be used in normal processes or in message event handlers. A message intermediate event receives a message in an already instantiated process. Because of this communicating behavior, these events can be connected with incoming message flows. A timer event is triggered by a specific deadline or duration and can be used for starting a timer event handler or as a delay within the process flow. A compensation event initiates the compensation of a specified process. An error event causes an error in the process it belongs to. Other event types, namely cancel, rule, link or multiple are not allowed to be used in BPMN$^+$, because they can not be transformed to BPEL4Chor.

Like BPMN, BPMN$^+$ allows attaching one or more intermediate events to the boundary of tasks and scopes. As explained in the BPMN specification [BPMN06] this creates an event context, which will respond to the specified trigger to interrupt the activity and redirect the flow through the attached event. This mechanism can be used to model error, compensation and termination handlers in BPMN$^+$. The actual handler is represented by a sub-process activity, which is connected with the attached event.

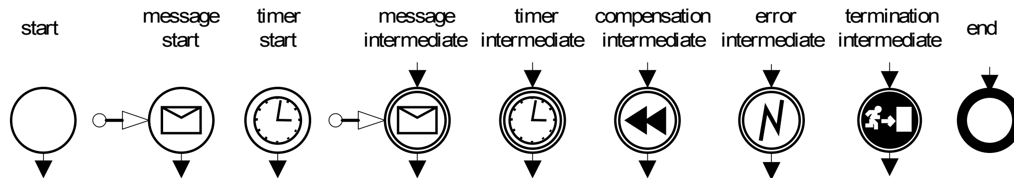


Figure 2.3: Events

To express termination handlers, a new type of event has to be introduced in BPMN⁺, the termination intermediate event. It can only be used as attached event. An outgoing sequence flow is used to connect it with the handler.

The event types, which can be used in BPMN⁺, are illustrated in Figure 2.3. The attributes of events can be found in section 2.3.17.

Task

A task can be of the type service, send, receive or none. In addition to this, BPMN⁺ introduces tasks of the type assign, empty and validate. The service, send and receive tasks can be connected with message flows (see section 2.2.6). Moreover tasks can be modeled as looping flow objects by marking them as standard or multi-instance loop.

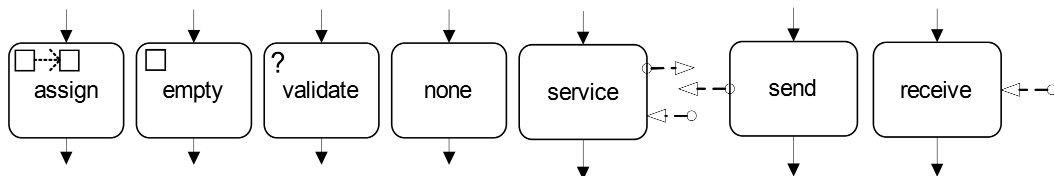


Figure 2.4: Task activities

Service tasks must be synchronous. This means they must be connected with incoming and outgoing message flows. The incoming message flows express the returned result a service task is waiting for to complete. Send tasks are only allowed to have outgoing message flows and receive tasks must have only incoming message flows.

An assign task expresses the assignment of values from one variable to another. Expressing the assignments using associations would be very complicated. The associations have to indicate which variable is the source, which one is the target and which data has to be copied. Moreover, there can be defined multiple assignments for one assign task and it has to be specified which of these assignments a variable belongs to. Such a bulk of associations is hard to understand. That is why the assignments are expressed in the attributes of an assign task and not using associations.

A validate task validates the content of variables against the variable definition. The variables to be evaluated are associated with the validate task. An empty task is a task that does nothing and a non-typed task does not specify what is actually done. To distinguish the assign, empty and validate tasks from each other in the diagram, special markers are introduced that can be seen in Figure 2.4.

Gateway

Gateways control the branching, forking, merging and joining of the sequence flow. In BPMN⁺ these are the only activities with more than one incoming or outgoing sequence flow. Not all types of gateways that are provided by BPMN are needed in BPMN⁺. Merely parallel fork, parallel join, data-based and event-based exclusive decision, exclusive

merge, inclusive decision and inclusive merge gateways are allowed. Other gateway types must not be used in BPMN$^+$, because they can not be transformed to BPEL4Chor.

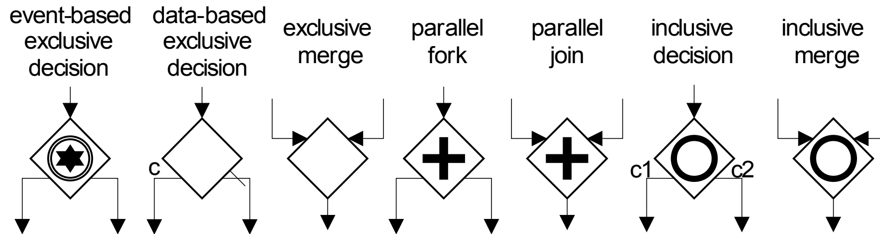


Figure 2.5: Gateway types

Sub-Process

In BPMN$^+$, a sub-process can be a scope or a handler and contains flow objects and the sequence flow between them.

If the sub-process is a scope, it has one incoming and one outgoing sequence flow. Like tasks, scopes can be marked as standard or multi-instance loops. The sequence flow within the scope can be started with the receipt of a message or with a non-triggered start event. Moreover, a scope is allowed to contain multiple message start events and multiple end events.

Handlers are introduced in BPMN$^+$ to simplify the modeling of event handling. They are the only flow objects that can be connected with attached intermediate events in the case of fault, compensation and termination handlers. In this way the activities that perform the event handling are more obvious and the restrictions that apply for these handlers can be validated more easily. Moreover, the marking of handlers with special markers leads to a more consistent representation, since in BPMN compensating sub-processes are already marked with markers.

Fault, compensation and termination handlers can contain multiple message start events and multiple end events. Message and timer event handlers must have a defined entry point. That is why they have to contain only one start event of a certain type. The process of a message event handler starts with a message start event and the process of a timer event handler starts with a timer start event.

Handlers are subject to restrictions regarding the connecting objects:

A fault handler is triggered using an attached error event. There must be an exception sequence flow from the attached error event to the fault handler. To represent a BPEL4Chor fault handler, the exception sequence flow has to be merged again with the normal sequence flow after the handler. This is done using an exclusive or inclusive merge gateway as can be seen in Figure 2.6. The merge has to be located directly after the fault handler and the faulted activity. In this way, the process flow will continue after the activity if no fault occurred or if an occurred fault was handled.

To associate a fault handler with the top-level process of a pool or pool set, an additional scope has to be modeled. The error event has to be attached to this scope, because BPMN does not allow attaching events to swimlane boundaries.

A compensation handler is not executed within the normal flow. That is why it cannot be connected with a sequence flow. It must have an incoming association that connects the handler with an attached compensation event.

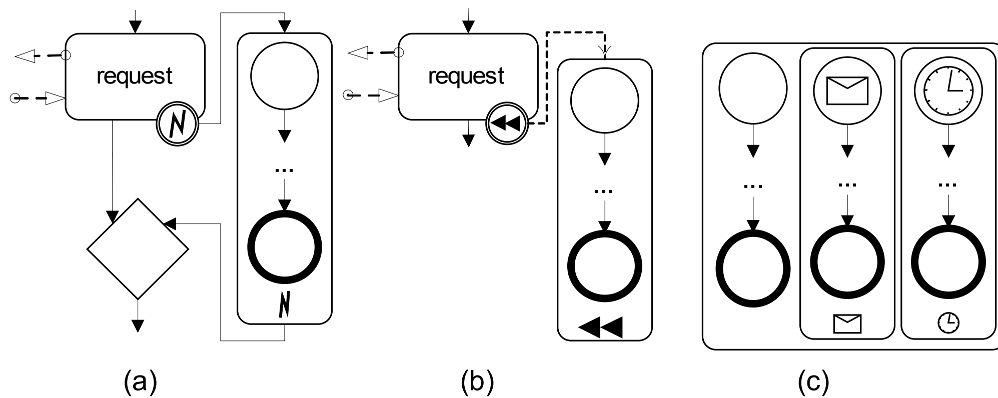


Figure 2.6: (a) fault handler and (b) compensation handler for synchronous service activity and (c) event handlers in scope

Message and timer event handlers cannot be expressed in a way that conforms to the BPMN specification. So we have to change the semantic of sub-processes to express these event handlers in BPMN$^+$. They are introduced as special sub-processes that can be instantiated by the receipt of a message or by an alarm. This instantiation is expressed using a message or a timer start event. Event handlers can be instantiated multiple times and run concurrently to the normal sequence flow. They can be located in a pool, pool set or sub-process and they can only be activated as long as the enclosing process or sub-process is active. This follows the semantic of BPEL4Chor event handlers.

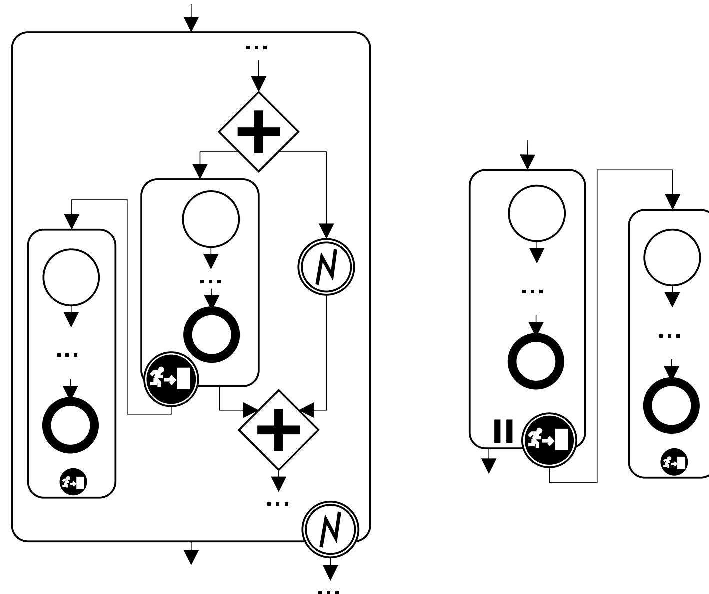


Figure 2.7: Termination handler within a scope or a parallel multi-instance loop

A termination event is introduced in BPMN$^+$ for handling the termination of activities. The actual control flow for handling the termination is contained in a termination handler that is introduced in BPMN$^+$, too. A termination handler is connected to an attached termination event using a sequence flow. This termination event is triggered when the enclosing scope fails or terminates. It is also triggered when the scope is a parallel multi-instance loop that is terminated by the completion condition of the loop. In the first case, the sequence flow continues after the faulted activity. Thus, there is no flow needed that leaves the termination handler. In the second case, the merge of each instance including

the termination handler is already part of the looping activity itself (defined by the MI_FlowCondition attribute). That is why there is no outgoing sequence flow needed for the termination handler as well. Figure 2.7 illustrates the usage of termination handlers in both cases. The semantic of termination events and termination handlers follows the BPEL4Chor semantic of termination handlers.

2.2.5 Artifacts

BPMN provides data objects, groups and text annotations for showing additional information about the BPD and its elements. In $BPMN^{+}$, only the data objects have a special semantical meaning that is important for modeling choreographies. Other artifact types can be used to structure the model or to provide additional information to the human reader.

Data objects can express variables containing data that is processed. Variable data objects must be located within the boundaries of a pool, pool set or sub-process and they can only be used from activities within those and their enclosed sub-processes. There are different variable types in $BPMN^{+}$. The default type is a standard variable that expresses the defined variables within a process or sub-process. A counter variable is used as counter within a multi-instance loop. This variable is located within the boundaries of the looping sub-process and marked with a special marker. The counter of a looping task does not need to be modeled explicitly. Despite of that, the counter of a looping task can be expressed with an association from the counter variable data object to the task. A fault variable holds the data of a fault that was caught. That is why it has to be associated with the appropriate attached error event. A message variable contains the message that triggers the message start event of a message event handler. That is why it has to be associated with the message start event. To distinguish the different variable data object types they are marked with special markers as shown in Figure 2.8.

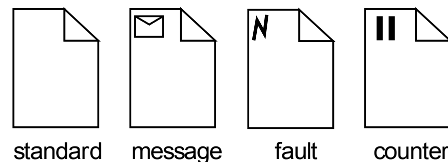


Figure 2.8: Standard, message, fault and counter variable data objects

Standard variable data objects can only be connected with service, receive, send and validate tasks. The variable handling of an assign task is described using attributes. Nevertheless, the used variables have to be modeled as data objects to provide the necessary information about the variable definitions.

In addition to variables, data objects are used for expressing the participants taking part in a conversation. Therefore participant reference and participant set data objects are introduced. A pool represents one participant of a certain type, whereas a pool set represents multiple participants of the same type. Participant data objects are only needed for conversations where multiple participants of the same type may take part in. This means that they are only needed, if communicating activities are located in a pool set. In this case the data objects are associated with the communicating activities.

The participants play an important role regarding to the modeling of choreographies. To distinguish them from variable data objects, new shapes are introduced, although this does not conform to the BPMN specification. These shapes can be seen in Figure 2.9. A participant reference or participant set is identified by the name of the data object. This

means that participant reference or participant set data objects with the same name represent the same participant reference or set respectively.

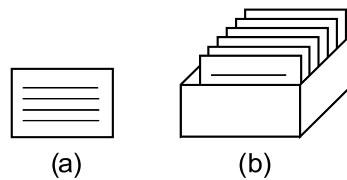


Figure 2.9: (a) participant reference and
(b) participant set data object

2.2.6 Connecting Objects

Message and sequence flow are connectors between flow objects. While the sequence flow expresses the sequential relationship between the flow objects of a process or sub-process, the message flow conveys messages between participants.

A message flow can only be connected with the following communicating flow objects:

Outgoing message flow:

- send task
- service task

Incoming message flow:

- message start event
- message intermediate event (not attached)
- receive task
- service task

The sequence flow always starts in a start event and ends in an end event of the same process or sub-process. It cannot cross the pool, pool set or sub-process boundaries. A start event has one outgoing sequence flow and an end event has one incoming sequence flow. Event handlers and compensation handlers have no incoming or outgoing sequence flow and termination handlers have only one incoming sequence flow. Attached intermediate events have only one outgoing sequence flow. The other flow objects have one incoming and one outgoing sequence flow. Gateways can have either multiple incoming or multiple outgoing sequence flows. For the attached fault events a special sequence flow called exception flow is used.

An association connects data objects with other graphical objects. If the association is not directed, it is used to connect a data object with a message flow. A directed association is used to connect variable data objects with tasks or events. This expresses the variables, which hold the data that is processed. The direction indicates the variable as input (arrowhead at the flow object) or output (arrowhead at the data object). An input variable means that a value is read from the variable (e.g. its value will be stored in a message) and an output variable means that a value is written to it (e.g. data of an incoming message). There are restrictions to the type of tasks and events that can be connected with variable data objects:

Association to variable data object only allowed for (output variable):

- message start event

- message intermediate event
- receive task
- error intermediate event (only attached)
- service task

Association from variable data object only allowed for (input variable):

- service task
- send task

In the case of an attached compensation intermediate event, BPMN provides a special association called compensation flow for connecting it with the compensation handler. This clarifies that the compensation flow is not a sequence flow because the attached compensation event can only be reached if the activity is already completed.

The direction of associations and the type of the connected elements are important for the semantic of participant reference and participant set data objects.

Association to communicating activities (see Figure 2.10):

(a) An association from a participant reference data object to a sending flow object means that a message is sent to this participant (participant reference is receiving the message)

(b) An association from a participant reference data object to a receiving flow object means that a message is expected from this participant (participant reference is sending the message).

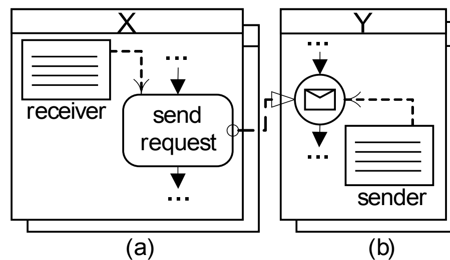


Figure 2.10: Associations to communicating activities

Association from communicating activities (see Figure 2.11):

(c) An association from a receiving flow object to a participant reference data object means that a message from an arbitrary participant is expected. The actual sender is bound to the participant reference.

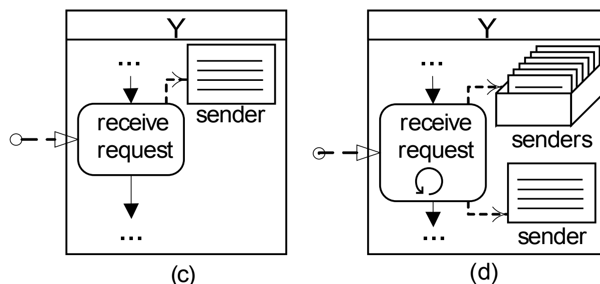


Figure 2.11: Associations from communicating activities

(d) An association from a receiving flow object to a participant set data object means that a message from an arbitrary participant is expected. The actual sender is added to the set if the sender is not already contained in it. A participant reference data object must be associated with the receiving flow to denote the participant reference the actual sender is bound to (see (c)).

Association to looping activities (see Figure 2.12):

(e) An association from a participant set data object to a multi-instance loop means that the loop is iterating over the participants contained in the set.

(f) An association from a participant reference data object to a multi-instance loop means that the participant reference acts as loop counter (participant set data object must be associated with the loop, too).

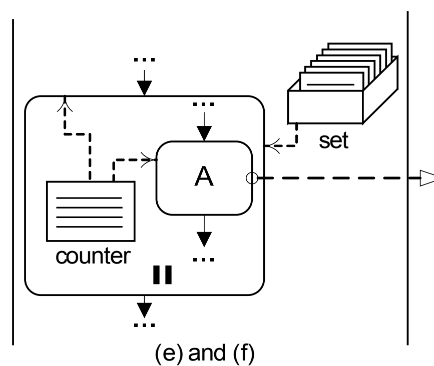


Figure 2.12: Participant data objects and multi-instance loops

Containment associations (see Figure 2.13):

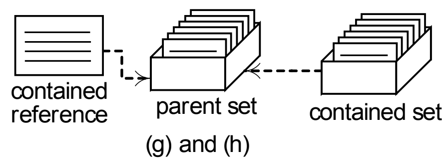


Figure 2.13: Containment associations for participant data objects

(g) An association from a participant reference to a participant set means that the participant reference is contained in the set.

(h) An association from a participant set to another participant set means that the first participant set is contained in the second one.

Passed participant references and sets (see Figure 2.14):

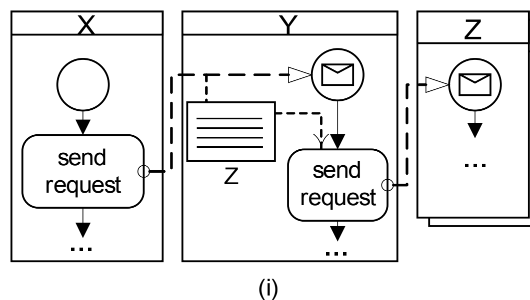


Figure 2.14: Participant reference passed over a message flow

(i) An undirected association from a participant reference or participant set data object to a message flow means that the participant reference or set is passed over the message flow.

2.3 Element attributes

Now that the graphical notation of the BPMN+ elements and their relationship among each other is defined, this section provides more details by specifying the attributes for each element. The tables below contain adapted BPMN attributes or additional attributes that do not occur in [BPMN06]. Those that are similar to the BPMN specification will not be listed. A complete set of attributes can be found in appendix A: BPMN+ Element Attributes.

The last column of each table indicates whether an attribute is unchanged, adapted or added to the list of attributes defined in BPMN. The attributes are marked as follows:

- Attribute introduced in BPMN+: [+]
- Attribute adapted in BPMN+: [*]
- Attribute used unchanged: [ ]

There is a special notation for defining the type of an attribute. Some attributes can have alternative values, which are separated by "|" and grouped within "(" and ")". To specify the default value of an attribute, it is separated from the actual type using a colon.

Notation for a type declaration: (alternative 1 | ... | alternative n) alternative i : type

2.3.1 Business Process Diagram

In BPMN+, a default value for the ExpressionLanguage and the QueryLanguage attribute of a BPD is defined. The default language for both attributes is XPath (see [XPath99]). Since BPMN+ introduces the pool set element, a pool set attribute is added. As a result, the pool attribute becomes optional.

Regarding the generation to BPEL4Chor the TargetNamespace attribute is introduced to identify the generated participant topology. The implementation details are decoupled from the choreography itself. That is why an additional GroundingFile attribute is introduced that references a file where the grounding is specified. In this way the grounding of the choreography is not fixed to a special format. Moreover, it can be exchanged easily to create different versions of the choreography with different groundings.

Attribute Name	Type	Description	
ExpressionLanguage (0-1)	"urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0" : String	See table 8.6 in [BPMN06] In BPMN+, the default language is XPath.	[*]
QueryLanguage (0-1)	"urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0" : String	See table 8.6 in [BPMN06] In BPMN+, the default language is XPath.	[*]
Pools (0-n)	Pool	See table 8.6 in [BPMN06] At least one pool or pool set has to be specified.	[*]

PoolSets (0-n)	Pool Set	A BPD can contain pool sets. At least one pool or pool set has to be specified.	[+]
TargetNamespace (0-1)	"urn:choreography/topology": String	This is the target namespace used for the participant topology.	[+]
GroundingFile (0-1)	String	Since the grounding will not be modeled within a BPMN$^+$ diagram, a grounding file can be associated with the choreography.	[+]

2.3.2 Process

BPMN$^+$ processes are mapped to BPEL abstract processes. That is why the ProcessType attribute can only have the value Abstract. The AdHoc attribute always has the value False, since BPMN$^+$ does not support ad hoc processes. That is why the related attributes for ad hoc processes are not mentioned in the table below. BPMN as well as BPMN$^+$ do not provide a mechanism to invoke compensation on the instance level of a process. That is why the EnableInstanceCompensation attribute always has the value False in BPMN$^+$.

The QueryLanguage and ExpressionLanguage attributes are introduced for a BPMN$^+$ process to overwrite the global query language and expression language defined for a BPD. The ExitOnStandardFault and MessageExchanges attributes are introduced for the mapping to BPEL4Chor.

Attribute Name	Type	Description	
ProcessType	Abstract : String	See table 8.7 in [BPMN06] BPEL4Chor uses abstract BPEL processes. Thus, the process type must be Abstract if the diagram will be transformed to BPEL4Chor.	[*]
AdHoc	False : Boolean	See table 8.7 in [BPMN06] BPMN$^+$ does not support ad hoc processes, so the value of this attribute is always False.	[*]
EnableInstanceCompensation	False : Boolean	See table 8.7 in [BPMN06] This attribute always has the value False in BPMN$^+$.	[*]
QueryLanguage (0-1)	String	This attribute value overwrites the QueryLanguage attribute value defined for the BPD.	[+]
Expression Language (0-1)	String	This attribute value overwrites the ExpressionLanguage attribute value defined for the BPD.	[+]
ExitOnStandardFault (0-1)	False : Boolean	If set to False the process can handle standard faults using fault handlers. If set to True, the process must exit immediately when a standard fault is encountered.	[+]
MessageExchanges	String	This attribute defines message exchange	[+]

(0-n)		names that can be used to disambiguate the relationship between inbound communicating flow objects and send tasks.	

2.3.3 Graphical Object

Like in BPMN, a graphical object is the common ground for all objects that are shown in the diagram. The attributes of a graphical object are not changed in BPMN$^+$.

2.3.4 Pool

A pool is a graphical object and inherits all its attributes. The Process attribute is not optional anymore in BPMN$^+$, because the process is essential in describing the behavior of a choreography participant. The Participant attribute of a pool is used to define the name of the participant, the pool represents. The type of the participant (role or entity) is irrelevant in BPMN$^+$. There may be other participant reference data objects in the diagram that have the same name like the participant specified for the pool.

In addition to the already existing Participant attribute, the Selects and Containment attributes are introduced for the mapping to a BPEL4Chor topology. The attribute values for TargetNamespace and Prefix can be used to reference elements located in the pool. The Imports attribute defines XML Schema or WSDL files that are used to declare variables located in the process of the pool.

Attribute Name	Type	Description	
Process	Process	See table 9.33 in [BPMN06] In BPMN$^+$, each pool must contain a process.	[*]
Participant	Participant	See table 9.33 in [BPMN06] The name of the Participant is the name of the participant reference the pool represents.	[*]
Selects (0-n)	String	This attribute defines the names of the participants that are selected by the participant reference this pool represents.	[+]
Containment (0-1)	(Required | MustAdd | AddIfNotExists) AddIfNotExists : String	This attribute describes the containment relationship of the participant reference represented by this pool in a participant set. This attribute must only be set if the participant reference is contained in a participant set.	[+]
TargetNamespace	"urn:choreography/process" : String	This is the target namespace of the process contained in the pool.	[+]
Prefix	"p": String	The prefix belongs to the target namespace above and it is used for referencing the process and the elements that are located in the pool.	[+]
Imports	Import	The import elements declare dependencies	[+]

(0-n)		on external XML Schema or WSDL definitions.	

2.3.5 Pool Set

A pool set is introduced with all its attributes in BPMN$^+$. A pool set is a graphical object and inherits all its attributes. These are nearly the same attributes as for a pool. However, a pool set does represent multiple participants of the choreography. These participants are expressed as participant reference or participant set data objects. So a pool set does not have the attributes necessary for describing a choreography participant.

Attribute Name	Type	Description	
Process	Process	This attribute defines the top-level process that is contained within the pool set. In BPMN$^+$, each pool set must contain a process.	[+]
Lanes (1-n)	Lane	Lanes do not add any semantical value in BPMN$^+$, but they can be used for layout purposes.	[+]
BoundaryVisible	True : Boolean	This attribute defines whether the boundary of the pool set is visible in the diagram.	[+]
TargetNamespace	"urn:choreography/process" : String	This is the target namespace of the process contained in the pool set.	[+]
Prefix	"p": String	The prefix belongs to the target namespace above and it is used for referencing the process and its elements that are located in the pool set.	[+]
Imports (0-n)	Import	The import elements declare dependencies on external XML Schema or WSDL definitions.	[+]

2.3.6 Lane

A lane is a graphical object and inherits all its attributes. With the introduction of a pool set in BPMN$^+$ the ParentPoolSet attribute has to be added for a lane. As a result, the ParentPool attribute becomes optional.

Attribute Name	Type	Description	
ParentPool (0-1)	Pool	See table 9.34 in [BPMN06] It must be specified either a parent pool or a parent pool set.	[*]
ParentPoolSet (0-1)	PoolSet	This is the parent pool set of the lane. It must be specified either a parent pool or a parent pool set.	[+]

2.3.7 Artifact

An artifact is a graphical object and inherits all its attributes. In addition to the pool attribute, a pool set attribute is specified to identify the location of the artifact.

Attribute Name	Type	Description	
PoolSet (0-1)	Pool Set	An artifact is either located in a pool, in a pool set or outside of those. This attribute specifies the pool set, the artifact is located in.	[+]

2.3.8 Annotation

The attributes of an annotation are not changed in BPMN$^+$.

2.3.9 Group

The attributes of a group are not changed in BPMN$^+$.

2.3.10 Data Object

A data object is a special artifact and inherits all artifact attributes. In BPMN$^+$ it expresses a variable, a participant reference or a participant set. The type of the data object is determined by the Type attribute introduced in BPMN$^+$.

Since not all data objects represent variables that are used by activities the ProducedAtCompletion attribute value depends on the type of the data object. A data object is only produced at the completion of an activity if the data object is used as output variable. In all other cases the value of this attribute is False. Moreover, the RequiredForStart attribute is always False in BPMN$^+$, because data objects do not block the start of an activity.

Attribute Name	Type	Description	
RequiredForStart	False : Boolean	See table 9.36 in [BPMN06] This attribute is always False in BPMN$^+$, since data objects do not block the start of an activity.	[*]
ProducedAtCompletion	False : Boolean	See table 9.36 in [BPMN06] The attribute value depends on the data object type in BPMN$^+$. A variable data object that is associated with an activity as output variable has this attribute set to True. For other data objects this attribute is set to False.	[*]
Type	(Variable | Reference | Set) : String	A data object in BPMN$^+$ can either represent a variable, a participant reference or a participant set.	[+]

Variable Data Object

A variable data object is introduced with all its attributes in BPMN$^+$. It is a special data object, thus it inherits all data object attributes. BPMN$^+$ knows counter, fault, message and standard variable data objects. The type of the data object is determined by the Va-

riableType attribute. The context of standard variable data objects can be restricted to a certain scope using the Scope attribute. To define the type of the variable data object, the DefinitionType, the DefinitionTypePrefix and the DefinitionTypeValue attributes are introduced. Since standard variable data objects can be initialized with a value, the VariableFromSpec attribute is used to specify this initialization.

Attribute Name	Type	Description	
VariableType	(Counter | Fault | Message | Standard) Standard : String	BPMN$^+$ introduces different data object types. A standard variable expresses the defined variables within a process or subprocess. The counter variable is used as counter within a multi-instance loop. The fault variable holds the data of a fault that was caught and the message variable contains the message that triggers a message event handler.	[+]
[VariableType = Standard] Scope (0-1)	Scope	A standard variable data object can be limited to a specified scope.	[+]
[VariableType = Standard, Message or Fault] DefinitionType	(MessageType | XMLType | XMLElement) : String	Variables can be defined in terms of WSDL message types, XML schema types or XML schema elements. If the VariableType is set to Fault or Message, this attribute must have the value MessageType or XMLElement.	[+]
[VariableType = Standard, Message or Fault] DefinitionTypePrefix	String	The prefix identifies the namespace of the imported file, the definition type of the variable data object is located in.	[+]
[VariableType = Standard, Message or Fault] DefinitionTypeValue	String	This attribute value has to match a WSDL message type, XML schema type or XML schema element defined in an imported file. If the VariableType is set to Fault, an XML schema type is not allowed.	[+]
[VariableType = Standard] VariableFromSpec (0-1)	FromSpec	With the FromSpec attribute a defined variable can be initialized. The attributes of a FromSpec can be found in section FromSpec.	[+]

Participant Reference Data Object

A participant reference data object is introduced with all its attributes in BPMN$^+$. It is a special data object, thus it inherits all data object attributes. A participant reference can select other participants that are taking part in the choreography. This selection is defined in the Selects attribute. If participant references are contained in a set they can define a containment relationship in the Containment attribute. Participant references that are pas-

sed over a message flow can define the participant reference they will be copied to in the CopyTo attribute. The context of a participant reference can be limited to a scope using the Scope attribute. This means that every time the scope is entered, there is no participant bound to the participant reference.

Attribute Name	Type	Description	
Selects (0-n)	String	This attribute defines the names of the participants that are selected by this participant reference.	[+]
Containment (0-1)	(Required | MustAdd | AddIfNotExists) AddIfNotExists : String	This attribute describes the containment relationship of the participant reference in a participant set. This attribute must only be set if the participant reference is contained in a participant set.	[+]
CopyTo (0-1)	String	If the participant reference data object is associated with a message flow the reference can be copied to another reference. The name of this reference is specified in the CopyTo attribute.	[+]
Scope (0-1)	Scope	The Scope attribute defines the scope the participant reference is limited to.	[+]

Participant Set Data Object

A participant set data object is introduced with all its attributes in BPMN^{+}. It is a special data object, thus it inherits all data object attributes. Participant sets that are passed over a message flow can determine the participant set they will be copied to in the CopyTo attribute. The context of a participant set can be limited to a scope using the Scope attribute. This means every time the scope is entered, there is no participant bound to the participant set.

Attribute Name	Type	Description	
CopyTo (0-1)	String	If the participant set data object is associated with a message flow the set can be copied to another set. The name of this set is specified in the CopyTo attribute.	[+]
Scope (0-1)	Scope	The Scope attribute defines the scope the participant set is limited to.	[+]

2.3.11 Connecting Object

A connecting object is a graphical object and inherits all its attributes. It is the base element for a sequence flow, a message flow and an association. To point out the restrictions of the source and target element types, these attributes are listed in the child elements. The other attributes of a connecting object do not change in BPMN^{+}.

2.3.12 Sequence Flow

The sequence flow is a connecting object and inherits all its attributes. It connects flow objects with each other. Because the sequence flow in BPMN^{+} generates only one token the Quantity attribute always has a value of 1.

Attribute Name	Type	Description	
Source	Flow Object	See table 10.1 in [BPMN06] An end event cannot be the source of a sequence flow.	[*]
Target	Flow Object	See table 10.1 in [BPMN06] A start event cannot be the target of a sequence flow.	[*]
Quantity	1 : Integer	See table 10.2 in [BPMN06] In BPMN^{+} this attribute is always 1.	[*]

2.3.13 Message Flow

A message flow is a connecting object and inherits all its attributes. In BPMN^{+}, the Message attribute is mandatory, because the name of the sent message must be known.

Attribute Name	Type	Description	
Source	Task	See table 10.1 in [BPMN06] This object can only be a service or a send task.	[*]
Target	Flow Object	See table 10.1 in [BPMN06] This activity can only be a service task, a receive task, a message start or a message intermediate event.	[*]
Message	Message	See table 10.3 in [BPMN06] In BPMN^{+}, this attribute is mandatory.	[*]

2.3.14 Association

An association is a connecting object and inherits all its attributes. In BPMN^{+}, the target of an association is always a data object. The source and the target elements of an association depend on the specified direction. An association with the direction Both is not used in BPMN^{+}.

Attribute Name	Type	Description	
Direction	(None | To | From) None : String	See table 10.4 in [BPMN06] In BPMN^{+}, the direction Both is not used.	[*]
Source	Object	See table 10.1 in [BPMN06] If the direction is None, the source has to be a message flow. If the direction is From, it has to be a message start event, a message intermediate event, an attached error intermediate event, a service task or an attached compensation event. If the direction is To, it has to be a send task, a service task or a compensation handler.	[*]

Target	Data Object	See table 10.1 in [BPMN06] If the source is an attached compensation event, the target must be a compensation handler. If the source is a compensation handler, the target must be an attached compensation event.	[*]

2.3.15 Flow Object

The flow object in BPMN$^+$ is similar to the flow object in BPMN. It is a graphical object and inherits all its attributes. The Pool attribute is optional in BPMN$^+$. In addition to this, a PoolSet attribute is introduced.

Attribute Name	Type	Description	
Pool (0-1)	Pool	See table 9.2 in [BPMN06] Each flow object is located either in a pool or in a pool set.	[*]
PoolSet (0-1)	Pool Set	This attribute identifies the pool set, the flow object belongs to. Each flow object is located either in a pool or in a pool set.	[+]

2.3.16 Gateway

As described in the BPMN specification, the gateway extends a flow object and inherits all its attributes. BPMN$^+$ supports only a subset of the types that BPMN provides for gateways. Therefore, the GatewayType attribute is restricted to exclusive, inclusive and parallel Gateways. In contrast to the BPMN specification the type Gate, which is used in the tables below, is added as supporting type in section 2.3.21 and not defined in the table itself.

Attribute Name	Type	Description	
GatewayType	(XOR | AND | OR) XOR : String	See table 9.25 in [BPMN06] In BPMN$^+$, only exclusive, parallel and inclusive gateways are used.	[*]

Parallel Gateway

Parallel fork and parallel join are gateways with the GatewayType attribute set to AND. In BPMN$^+$, each process ends with an end event. Thus, gateways are not allowed to be the last object in the sequence flow and have to define at least one gate.

Attribute Name	Type	Description	
Gates (1-n)	Gate	See table 9.30 in [BPMN06] In BPMN$^+$, a parallel gateway must have at least one outgoing gate.	[*]

Exclusive Gateway

Data-based exclusive decision, event-based exclusive decision and exclusive merge are gateways with the GatewayType attribute set to XOR. Like a parallel gateway, a data-based exclusive gateway must have at least one gate defined.

Attribute Name	Type	Description	
[XORType = Data] Gates (1-n)	Gate	See table 9.26 in [BPMN06] In BPMN+, an exclusive gateway must have at least one outgoing gate.	[*]

Inclusive Gateway

Inclusive decision and merge gateways are gateways with the GatewayType attribute set to OR. Like the parallel gateway, an inclusive gateway must have at least one gate defined.

Attribute Name	Type	Description	
Gates (1-n)	Gate	See table 9.28 in [BPMN06] In BPMN+, an inclusive gateway must have at least one outgoing gate.	[*]

2.3.17 Event

As described in the BPMN specification, an event extends a flow object and inherits all its attributes. The attributes common to all events are not changed in BPMN+.

Start Event

BPMN+ supports only a subset of the start event types. Thus, the Trigger attribute is restricted to None, Message and Timer.

The message, which has to be supplied to a message start event, is conveyed by the message flow connected to the start event. That is why it does not need to be specified explicitly. The attributes MessageExchange, Correlations are introduced for the mapping of message start event to BPEL4Chor. The FromParts or OpaqueOutput attribute can be used instead of associating an output variable with the event. Each FromPart defines in which variable a certain part of the received message should be stored. The OpaqueOutput attribute specifies that the actual output variable is hidden.

The TimeType attribute is needed to decide if the value of the TimeCycle or the TimeDate value should be specified. Without this attribute it could not be determined whether an empty attribute value means that it is hidden or not set. The TimeLanguage attribute is introduced in addition to identify the language that is used for the TimeCycle or TimeDate attribute. In BPMN+, a timer start event can be triggered repeatedly. For this purpose the RepeatEvery attribute is introduced.

Attribute Name	Type	Description	
Trigger	(None | Message | Timer) None : String	See table 9.5 in [BPMN06] In BPMN+, only start events with the trigger None, Message and Timer are used.	[*]

[Trigger = Message] Message	Message	See table 9.5 in [BPMN06] In $BPMN^+$, the message is already provided by the message flow connected to the start event.	[*]
[Trigger = Message] MessageExchange (0-1)	String	This attribute defines a message exchange name that can be used to disambiguate the relationship between this event and the following send tasks.	[+]
[Trigger = Message] Correlations (0-n)	Correlation	The Correlations attribute is used to identify a conversation. The attributes of a correlation can be found in 2.3.21.	[+]
[Trigger = Message] OpaqueOutput	True: Boolean	The output variable can be opaque to hide the actual data that is received. If this attribute is True, the FromParts must not be specified and there must not be an output variable associated with this event.	[+]
[Trigger = Message] FromParts (0-n)	FromPart	The FromParts can be used instead of modeling an association to the data object containing the received Message.	[+]
[Trigger = Timer] TimeType	(Date | Cycle) Date : String	This attribute is necessary to distinguish opaque from not used time definitions.	[+]
[Trigger = Timer] TimeLanguage (0-1)	String	Defines the language that is used to express the TimeCycle or the TimeDate value.	[+]
[Trigger = Timer] RepeatEvery (0-1)	Expression	The RepeatEvery attribute holds a duration expression that defines the interval in which the event will be triggered repeatedly.	[+]

End Event

$BPMN^+$ only knows end events with the result type None and there are no attributes defined for end events with this type.

Attribute Name	Type	Description	
Result	None: String	See table 9.7 in [BPMN06] There are only end events with a Result of None in $BPMN^+$.	[*]

Intermediate Event

$BPMN^+$ uses message, timer, error and compensation intermediate events. Moreover, every intermediate event has to specify a trigger, so None is not a valid value for the Trigger attribute. In addition to this, the trigger type "Termination" is introduced that is used to express the termination handler of a task or sub-process.

Intermediate events can be attached to the boundaries of activities in BPMN. This is only allowed for compensation, termination and error events in BPMN$^+$. They can be attached to the boundaries of scopes or tasks.

The message, which has to be supplied to a message intermediate event, is conveyed by the message flow connected to the event. That is why it does not need to be specified explicitly. The attributes MessageExchange and Correlations are introduced for the mapping of intermediate events to BPEL4Chor. As for the message start event, the FromParts, OpaqueOutput, TimeType and TimeLanguage attributes are introduced for intermediate events in BPMN$^+$.

BPMN$^+$ allows the "rethrowing" of errors using an error intermediate event within a fault handler that does not define an error code. That is why the ErrorCode attribute is optional. Also the Activity attribute of compensation intermediate events is optional. If this attribute is not defined all completed inner scopes will be compensated.

Attribute Name	Type	Description	
Trigger	(Message | Timer | Error | Compensation | Termination) Message : String	See table 9.9 in [BPMN06] In BPMN$^+$, only intermediate events with the trigger Message, Timer, Error, Compensation and Termination are used.	[*]
SuppressJoinFailure	False: Boolean	This attribute determines if join failures should be suppressed when there are multiple paths that lead to the event.	[+]
[Trigger = Error, Compensation or Termination] Target (0-1)	Object	See table 9.9 in [BPMN06] In BPMN$^+$, only intermediate events with the trigger Error, Compensation and Termination can be attached.	[*]
[Trigger = Message] Message	Message	See table 9.9 in [BPMN06] In BPMN$^+$, the message is already provided by the message flow connected to the event.	[*]
[Trigger = Message] MessageExchange (0-1)	String	This attribute defines a message exchange name that can be used to disambiguate the relationship between the message event and the following send tasks.	[+]
[Trigger = Message or Error] FromParts (0-n)	FromPart	If there is an incoming message flow connected with to event, the FromParts can be set. This is used instead of modeling an association to the variable data object containing the received message data.	[+]
[Trigger = Message] OpaqueOutput	True: Boolean	The output variable can be opaque to hide the actual data that is received. If this attribute is True, the FromParts must not be specified and there must not be an output variable associated with this event.	[+]

[Trigger = Message] Correlations (0-n)	Correlation	The Correlations attribute is used to identify a conversation. The attributes for a Correlation can be found in 2.3.21.	[+]
[Trigger = Timer] TimeType	(Date | Cycle) Date : String	This attribute is necessary to distinguish opaque from non-used time expressions.	[+]
[Trigger = Timer] TimeLanguage (0-1)	String	Defines the language that is used to express the TimeCycle or the TimeDate value.	[+]
[Trigger = Error] ErrorCode (0-1)	String	See table 9.9 in [BPMN06] The ErrorCode attribute can be omitted if the event is located in a fault handler. In this case the error, which was caught in the fault handler, will be rethrown.	[*]
[Trigger = Compensation] Activity (0-1)	Object	See table 9.9 in [BPMN06] In BPMN+ this attribute is optional. If this attribute is not defined, all successfully completed inner scopes will be compensated.	[*]

2.3.18 Activity

An activity inherits all attributes of a flow object. The InputSets and OutputSets attribute is not needed in BPMN+, because the input and output data is expressed using data objects. Since there is no relation between this input and output date, the IORules attribute is not needed as well.

In BPMN+, each activity starts as soon as a token arrives. That is why the StartQuantity attribute always has a value of “1”. A LoopType attribute value different from None is only allowed for tasks and scopes in BPMN+. The SuppressJoinFailure attribute is introduced for the mapping to BPEL4Chor.

Attribute Name	Type	Description	
StartQuantity	1 : Integer	See table 9.10 in [BPMN06] This attribute is always 1 in BPMN+.	[*]
LoopType	(None | Standard | MultiInstance) None : String	See table 9.10 in [BPMN06] Since only scopes and tasks can be looping activities in BPMN+, the attribute is always None for the other activity types.	[*]
SuppressJoinFailure	False: Boolean	This attribute determines whether join failures should be suppressed when there are multiple paths leading to the activity.	[+]

Loop

Looping activities can define certain attributes, depending on the loop type.

The StartCounterValue, FinalCounterValue, CompletionCondition and SuccessfulBranchesOnly attributes are introduced for the mapping to BPEL4Chor. In $BPMN^{+}$, a looping activity only completes if all its branches have completed or if the resulting branches were terminated by the CompletionCondition. That is why the MI_FlowCondition always has the value All. The loop counter is expressed with a counter variable data object, so the LoopCounter attribute is not needed in $BPMN^{+}$. Moreover, the definition of a LoopMaximum for standard loops is not allowed in BPMN. For multi-instance loops the MI_Condition attribute is replaced by the StartCounterValue, FinalCounterValue and CompletionCondition attributes, which determine the number of times the activity will be repeated.

Attribute Name	Type	Description	
[MI_Ordering = Parallel] MI_FlowCondition	(None | One | All | Complex) All : String	See table 9.12 in [BPMN06] This attribute is always set to All in $BPMN^{+}$.	[*]
[LoopType = Multi-Instance] StartCounterValue (0-1)	Expression	This is an unsigned-integer expression the counter will be initialized with. If a participant set data object is associated with the loop, this attribute must not be specified.	[+]
[LoopType = Multi-Instance] FinalCounterValue (0-1)	Expression	This is an unsigned-integer expression that defines the final counter value. If a participant set data object is associated with the loop, this attribute must not be specified.	[+]
[LoopType = Multi-Instance] CompletionCondition (0-1)	Expression	This is an unsigned-integer expression that allows completing without executing or finishing all branches specified. If a participant set data object is associated with the loop, this attribute must not be specified.	[+]
[LoopType = Multi-Instance] SuccessfulBranchesOnly (0-1)	False : Boolean	This attribute can only be specified if the CompletionCondition attribute is set. If set to True, only branches which have completed successfully must be counted. If set to False, all branches must be counted	[+]

2.3.19 Task

A task inherits all attributes from an activity. There is already a TaskType attribute defined in BPMN, but the task types need to be extended in $BPMN^{+}$. That is why the task types Assign, Empty and Validate are added. The BPMN task types User, Script, Manual and Reference are not allowed.

Attribute Name	Type	Description	
TaskType	(Service | Receive | Send | Assign, Empty | Validate | None) None :	See table 9.17 in [BPMN06] Only tasks of the type Service, Receive, Send, Assign, Empty, Validate and None are allowed in $BPMN^{+}$. The task of type Assign, Empty or Validate are marked	[*]

	String	with a special marker.	

Service Task

A service task inherits all attributes of a task. The InMessage and OutMessage are already provided by the message flows connected with the task. That is why they do not need to be specified explicitly. The Correlations attribute is introduced for the mapping to BPEL4Chor. The ToParts or OpaqueInput attribute can be used instead of associating an input variable with the task. A ToPart defines in which part of the message the value of a certain variable should be stored. The FromParts or OpaqueOutput attribute can be used instead of associating an output variable with the task. A FromPart defines in which variable a certain part of the received message should be stored.

Attribute Name	Type	Description	
InMessage	Message	See table 9.18 in [BPMN06] In BPMN$^+$, the message is already provided by the outgoing message flow connected to the task.	[*]
OutMessage	Message	See table 9.18 in [BPMN06] In BPMN$^+$, the message is already provided by incoming message flow connected to the task.	[*]
Correlations (0-n)	Correlation	The Correlations attribute is used to identify a conversation.	[+]
OpaqueInput	True: Boolean	The input variable can be opaque to hide the actual data that is sent. If this attribute is True, the ToParts must not be specified and there must not be an input variable associated with this task.	[+]
ToParts (0-n)	ToPart	The ToParts can be used instead of modeling an association from a data object which is the input variable.	[+]
OpaqueOutput	True: Boolean	The output variable can be opaque to hide the actual data that is received. If this attribute is True, the FromParts must not be specified and there must not be an output variable associated with this task.	[+]
FromParts (0-n)	FromPart	The FromParts can be used instead of modeling an association to a data object which is the output variable.	[+]

Receive Task

A receive task inherits all attributes of a task. As for the service task, the Message attribute does not need to be specified explicitly. The Correlations and MessageExchange attributes are introduced for the mapping to BPEL4Chor. Similar to the service task, the FromParts or OpaqueOutput attribute can be used instead of associating an output variable with the task.

Attribute Name	Type	Description	
Message	Message	See table 9.19 in [BPMN06] In BPMN+, the message is already provided by the incoming message flow connected to the task.	[*]
Correlations (0-n)	Correlation	The Correlations attribute is used to identify a conversation.	[+]
OpaqueOutput	True: Boolean	The output variable can be opaque to hide the actual data that is received. If this attribute is True, the FromParts must not be specified and there must not be an output variable associated with this task.	[+]
FromParts (0-n)	FromPart	The FromParts can be used instead of modeling an association to a data object which is the output variable.	[+]
MessageExchange (0-1)	String	This attribute defines a message exchange name that can be used to disambiguate the relationship between the receive task and following send tasks.	[+]

Send Task

A send task inherits all attributes of a task. As for the service task, the Message attribute does not need to be specified explicitly. The Correlations and MessageExchange attributes are introduced for the mapping to BPEL4Chor. In BPMN+, a send task can indicate that an error is replied. For this purpose a FaultName attribute can be specified. Similar to the service task, the ToParts or OpaqueInput attribute can be used instead of associating an input variable with the task.

Attribute Name	Type	Description	
Message	Message	See table 9.20 in [BPMN06] In BPMN+, the message is already provided by the outgoing message flow connected to the task.	[*]
Correlations (0-n)	Correlation	The Correlations attribute is used to identify a conversation.	[+]
FaultName (0-1)	String	This attribute can be specified to indicate that a fault message is replied. The fault name can only be specified if the send task is connected with a service task by a message flow.	[+]
MessageExchange (0-1)	String	This attribute defines a message exchange name that can be used to disambiguate the relationship between receiving activities and the send task.	[+]
OpaqueInput	True: Boolean	The input variable can be opaque to hide the actual data that is replied. If this attri-	[+]

		bute is True, the ToParts must not be specified and there must not be an input variable associated with this task.	
ToParts (0-n)	ToPart	The ToParts can be used instead of modeling an association from a data object which is the input variable.	[+]

Assign Task

The assign task inherits all attributes of a task. Since an assign task does not exist in BPMN, the attributes below are introduced in BPMN$^+$.

Attribute Name	Type	Description	
Validate	False : Boolean	The validate attribute specifies to validate all variables modified by the Copy attribute against their definition.	[+]
Copy (1-n)	Copy	This attribute is used to copy data from one variable to another, as well as to construct and insert new data using expressions. The attributes of a Copy type are listed in 2.3.21.	[+]

Empty Task

The empty task inherits all attributes of a task, but it has no additional attributes.

Validate Task

The validate task inherits all attributes of a task, but it has no additional attributes.

2.3.20 Sub-Process

A sub-process inherits all attributes of an activity. There is already a SubProcessType attribute defined in BPMN, but the type of a sub-process is always embedded in BPMN$^+$. In this way the activities connected with message flows can always be seen in the diagram. There are different embedded sub-process types needed in BPMN$^+$. That is why the attribute EmbeddedSubProcessType is introduced. BPMN$^+$ does not support transactions or ad hoc processes, so the IsATransaction and the AdHoc attributes always have the value False.

Attribute Name	Type	Description	
SubProcessType	(Embedded | Independent | Reference) Embedded : String	See table 9.13 in [BPMN06] This attribute always has the value Embedded in BPMN$^+$.	[*]
EmbeddedSubProcessType	(Scope | Handler) Scope : String	BPMN$^+$ introduces the new sub-process types Scope and Handler.	[+]
IsATransaction	False : Boolean	See table 9.13 in [BPMN06] This attribute is always False in BPMN$^+$.	[*]
[SubProcessType =	False: Boolean	See table 9.14 in [BPMN06]	[*]

Embedded] AdHoc		BPMN+ does not support ad hoc sub-processes, so the value of this attribute is always False.	

Scope

A scope is a special sub-process and inherits all its attributes. The attributes ExitOnStandardFault, MessageExchanges and Isolated are introduced in BPMN+ for the mapping to BPEL4Chor.

Attribute Name	Type	Description	
ExitOnStandard-Fault (0-1)	False : Boolean	If set to False, the process within the scope can handle standard faults using fault handler. If set to True, the process must exit immediately when a standard fault is encountered.	[+]
MessageExchanges (0-n)	String	This attribute defines message exchange names that can be used to disambiguate the relationship between inbound communicating flow objects and send tasks.	[+]
Isolated (0-1)	False : Boolean	If set to True, this attribute provides control of concurrent access to shared resources.	[+]

Handler

Handlers are special sub-processes and inherit all its attributes. BPMN+ knows fault, message, timer, termination and compensation handlers. The parent attribute of message or timer handlers determines the pool, pool set or sub-process the handler is located in. The attributes ExitOnStandardFault and Isolated were introduced for the mapping to BPEL4Chor.

Attribute Name	Type	Description	
HandlerType	(Fault | Message | Timer | Termination | Compensation) : String	The handler type of fault, termination and compensation handlers is determined by the attached intermediate event, the handler is connected with. A sequence flow with an attached error intermediate event leads to a fault handler. A sequence flow with an attached termination event leads to a termination handler. A compensation handler is not allowed to have any incoming or outgoing sequence flow. It is associated with an attached compensation intermediate event. Message and timer handlers are not allowed to have incoming or outgoing sequence flow. The type of these handlers is determined by the contained start event. A timer handler contains a timer start event and a message handler contains a message	[+]

		start event.	
[HandlerType = Message or Timer] Parent	Object	This is the name of the parent process or sub-process, the handler is located in.	[+]
[HandlerType = Message or Timer] ExitOnStandard-Fault (0-1)	False : Boolean	If set to False, the process within the handler can handle standard faults using fault handlers. If set to True, the process must exit immediately when a standard fault is encountered.	[+]
[HandlerType = Message or Timer] Isolated (0-1)	False : Boolean	If set to True, this attribute provides control of concurrent access to shared resources.	[+]

2.3.21 Supporting Types

For the new attributes, new types were introduced that will be specified in detail in the next tables. The supporting types from [BPMN06] will only be listed in this section, if they are adapted in BPMN$^+$.

Import

The Import type is used to declare process dependencies on external XML schema or WSDL files. The definitions contained in these files can be used in the variable declarations.

Since the concrete variable types are not defined in the early development stages the Location attribute is optional. In this way the files do not need to be created at the start of the process development. Moreover, they can be exchanged easily to design different versions of the choreography with different variable types.

Attribute Name	Type	Description	
Namespace	String	This attribute identifies the definitions of an imported file.	[+]
Location (0-1)	String	This attribute specifies the location of the imported file with the relevant definitions.	[+]
ImportType	String	This attribute identifies the type of document being imported. The type is specified by an URI that identifies the encoding language used in the document (e.g. “http://www.w3.org/2001/XMLSchema” or “http://schemas.xmlsoap.org/wsdl/”).	[+]
Prefix	String	The Prefix is used to identify the namespace of the definitions.	[+]

Copy

The Copy type is used to copy data from one variable to another, as well as to construct and insert new data using expressions.

Attributes		Description	
KeepSrcElementName (0-1)	False : Boolean	This attribute specifies whether the name of the destination will be replaced by the name of the source.	[+]
FromSpec	FromSpec	The FromSpec specifies the source of the copy operation.	[+]
ToSpec	ToSpec	The ToSpec specifies the target of the copy operation.	[+]

FromSpec

The FromSpec type is used to select a value from a variable, property, expression or literal.

Attribute	Type	Description	
FromSpecType	(Variable| VarProperty | Expression | Literal | Opaque | Empty) Empty : String	This determines the type of the FromSpec. The value can be selected from a variable, a variable property, an expression or a literal value. The type Opaque means that the actual value selection is hidden. The Empty type means that the FromSpec is empty.	[+]
[FromSpecType = Variable or VarProperty] VariableName	String	The name of a defined variable that the value will be selected from.	[+]
[FromSpecType = Variable] Part (0-1)	String	This attribute denotes the part name containing the value to select. It must match an existing part name of the variable. Thus, it is only set if the variable has the type MessageType.	[+]
[FromSpecType = Variable] Query (0-1)	Query	Optionally a query can be specified to select a value from the source variable or a message part.	[+]
[FromSpecType = VarProperty] Property	String	This attribute denotes the property name containing the value to be assigned to the variable. It must match an existing property defined for the variable.	[+]
[FromSpecType = Expression] Expression	Expression	This attribute defines the expression that returns a value.	[+]
[FromSpecType = Literal] Literal	String	This attribute defines a literal value that will be assigned to the variable.	[+]

ToSpec

The ToSpec type is used to define the target of a value assignment.

Attributes		Description	
ToSpecType	(Variable| Var-Property | Ex-pression | Em-pty) Empty : String	This determines the type of the ToSpec. The target of the assignment can be a variable, a variable property or an expression. The Empty type means, the ToSpec is empty.	[+]
[ToSpecType = Variable or VarProperty] VariableName	String	The name of a target variable.	[+]
[ToSpecType = Variable] Part (0-1)	String	This attribute denotes the target part name that the value will be assigned to. It must match an existing part name for the variable. Thus, it is only set if the variable has the type MessageType.	[+]
[ToSpecType = Variable] Query (0-1)	Query	Optionally a query can be specified to select a value from the target variable or a message part.	[+]
[ToSpecType = VarProperty] Property	String	This attribute denotes the property name the value will be assigned to. It must match an existing property defined for the variable.	[+]
[ToSpecType = Expression] Expression	Expression	This attribute defines the expression to select a value.	[+]

Query

The Query type is used to select a value from a source variable, a target variable or a message part.

Attributes		Description	
Language (0-1)	String	This attribute specifies the language of the query.	[+]
Content (0-1)	String	The Content attribute defines the expression to select a value. If no content is specified, the query is supposed to be opaque.	[+]

Correlation

The Correlation type is used to identify a conversation.

Attributes		Description	
Set	String	This attribute must match a correlation set name that is specified in the enclosing process.	[+]
Initiate	(Yes | No | Join) No : String	The Initiate attribute determines if the related activity must attempt to initiate the corresponding correlation set. If set to Join, it must only attempt to initiate the correlation set if it is not yet initiated.	[+]
Pattern (0-1)	(Request | Response | Request-Response) : String	The Pattern attribute must be specified if the correlation belongs to a task of type service. It is used to indicate whether the correlation applies to the request message, the response message or both.	[+]

FromPart

The FromPart type is used to retrieve data from an incoming message and place it into a variable.

Attributes		Description	
Part	String	The Part attribute references the part of the message that the value will be taken from. The value “##opaque” means to hide the actual part.	[+]
ToVariable	String	The ToVariable indicates the variable name that the value will be copied to. The value “##opaque” means to hide the actual variable.	[+]

ToPart

The ToPart type is used to create an anonymous temporary variable and assign the value from another variable to a part of it.

Attributes		Description	
Part	String	The Part attribute references the part of the anonymous temporary variable the variable value will be copied to. The value “##opaque” means to hide the actual part.	[+]
FromVariable	String	The FromVariable indicates the variable name the value will be copied from. The value “##opaque” means to hide the actual variable.	[+]

Expression

The type Expression is already defined in BPMN, but there is the additional attribute ExpressionLanguage needed in $BPMN^+$. Furthermore, the Expression attribute is changed to an optional attribute.

Attributes		Description	
ExpressionLanguage (0-1)	String	This specifies the language for the expression.	[+]
Expression (0-1)	String	See table B.43 in [BPMN06] In BPMN$^+$, this attribute is optional. If this attribute is not set, the expression interpreted as opaque.	[*]

Gate

The type Gate is already defined in BPMN within the declaration of the gateway attributes. In BPMN$^+$, it is declared as supporting type. Moreover, a gate in BPMN has an Assignments attribute, which is not needed in BPMN$^+$.

Attributes		Description	
OutgoingSequence-Flow	SequenceFlow	This is the sequence flow associated with the gate. Its Condition attribute depends on the gateway, the gate belongs to.	[*]

2.4 BPMN$^+$ Example

In this section the basic structure of a BPMN$^+$ diagram is explained using the example from chapter 1.1. In Figure 2.15 the diagram of Figure 1.1 is extended with the elements introduced in BPMN$^+$.

As it can be seen, the participant types Client and Bookshop are illustrated as Pools in the diagram. The various Suppliers are represented as a pool set. The name of the participant reference represented by a pool is defined as pool attribute. The pool and pool set names represent the names of the participant types.

The client invokes the bookshop process by sending a request message over the message flow. Since there are no multiple participants of the same type taking part in the communication, participant data objects are not associated with the communicating activities. After the invocation, the client waits for the response from the bookshop. This is modeled with an event-based exclusive gateway. The message intermediate events determine the paths that will be taken. The events are triggered by the receipt of the appropriate response message from the bookshop. To merge these branches, an exclusive join gateway is used that leads to the end of the client process. There are no variable data objects used in the client process, because the process ends after the receipt and the received messages do not need to be stored for further processing.

The usage of variable data objects can be seen in the bookshop process. The received fault data is stored in the “not found” fault variable. To define the type of this variable data object, the WSDL message type must be defined in an imported WSDL file. The import is specified as attribute of the bookshop pool. The definition is referenced in the fault variable data object using the defined import prefix.

The service task that invokes the process of a supplier is contained in a parallel multi-instance loop. To express that the loop is iterating over all known suppliers, the participant set data object “suppliers” is associated with the loop. Moreover, the participant reference data object “current supplier” acts as loop counter and is associated with the loop, too. In this way the current supplier of the loop instance is be bound to the “current supplier” data object. The invocation is synchronous and the response of the supplier can eit-

her be a message with the price or a fault message. The fault “notFound” is caught using the fault handler modeled with the attached error intermediate event. In the fault handler, a simple opaque activity is invoked that updates the supplier data of the bookshop. Details about the update are not part of the process. After the fault handler has completed the exception flow is merged again with the normal sequence flow using the exclusive merge gateway. When the responses from each supplier are received, a data-based parallel gateway is used to decide either to determine the cheapest supplier or to reply a fault message to the client. The exclusive merge gateway is used to join the branches and leads to the end of the bookshop process.

Each supplier process is initiated by the receipt of the price request from the bookshop. Since the suppliers are contained in a participant set, the actual requested supplier is expressed as participant reference data object associated with the service task. Based on the request, the supplier calculates his price of the book. The calculation of the price is hidden in a non-typed task. After that, a data-based decision gateway is used to decide whether a fault or the price will be replied to the bookshop. An exclusive merge gateway is used to join the branches and leads to the end of the supplier’s process. Variables used in this process are opaque. That is why there is no variable data object shown.

The element attributes are shown as text annotations in the diagram. This is not part of a typical BPMN+ diagram, because the attributes will be entered using an editor. Only mandatory attributes are illustrated that cannot be determined from the diagram and that do not have a default value. In addition to this, the Id attribute is omitted because it is typically generated by an editor. The WSDL file imported for the bookshop pool set is not given, too. However, it must define a fault message type “notFound” that is used by the fault variable data object. There is no grounding file attribute specified, because the purpose of this example is to clarify the use of BPMN+ and not of BPEL4Chor.

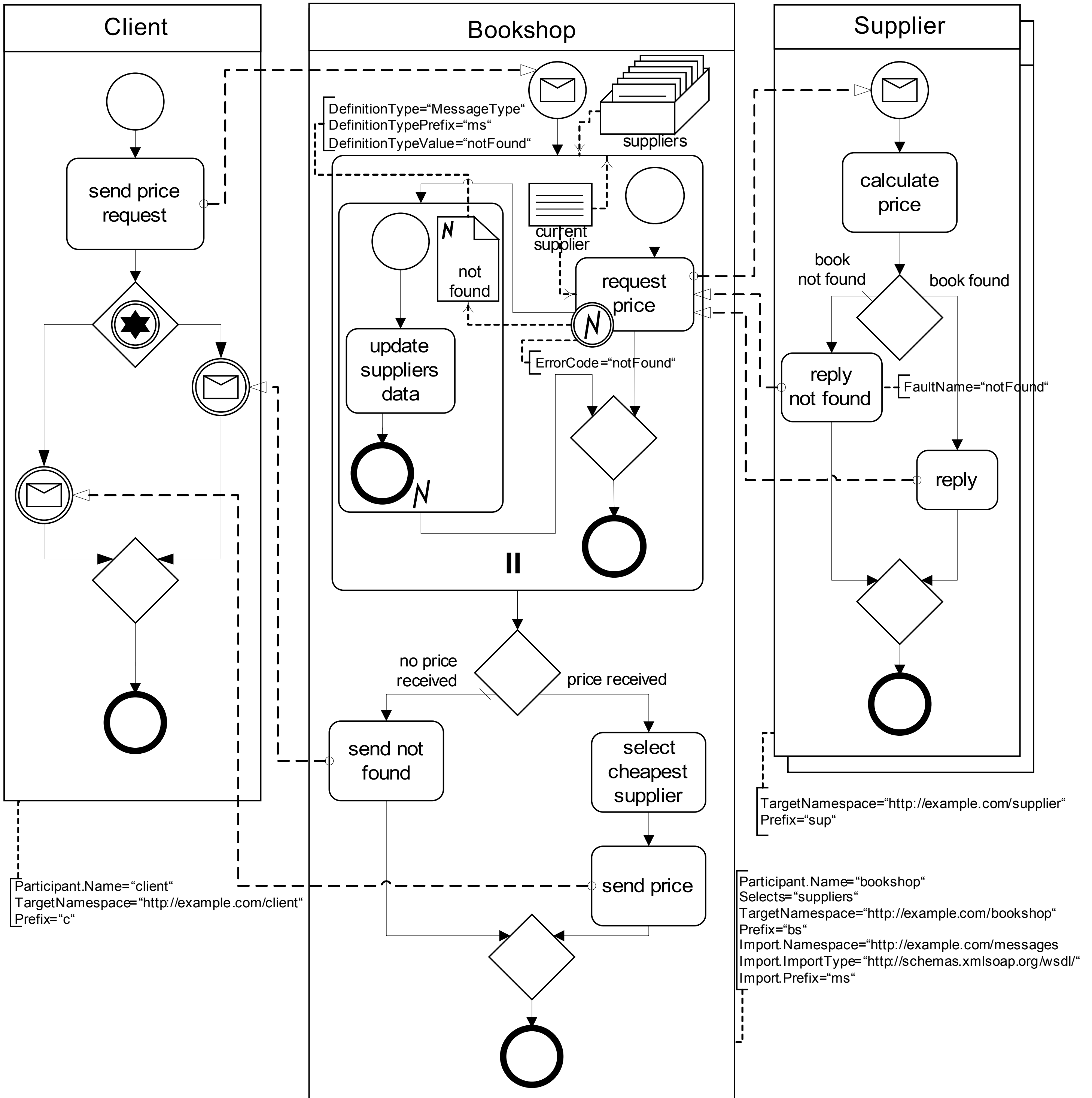


Figure 2.15: Requesting the price of a book in BPMN+

3 BPMN+ to XPDL4Chor

In the previous chapter the meta-model for BPMN+ was introduced. This meta-model extends BPMN with elements and attributes that are needed for modeling choreographies in the terms of BPEL4Chor. One of the goals of this thesis is to generate BPEL4Chor from BPMN+ diagrams that were modeled in a graphical editor. The transformation should be independent from this editor. For this purpose an intermediate format is needed to store and exchange the diagram between the editor and the transformation. BPMN does not provide an interchange format in its specification. However, the XML process definition language (XPDL, see [XPDL05]) is intended to be a file format for BPMN. In [Palm06] the notion of a "BPMN-XPDL-BPEL" value chain is described and in the next two chapters this concept will be extended to a "BPMN+-XPDL4Chor-BPEL4Chor" value chain.

This section describes the first step from BPMN+ to XPDL4Chor. The extensions made in BPMN+ must be reflected in XPDL. So first XPDL will be investigated regarding its extensibility. After that the representation of BPMN+ in XPDL will be described and XPDL will be extended where necessary. This extension is called XPDL4Chor.

3.1 XPDL Analysis

XPDL is a format standardized by the Workflow Management Coalition (WfMC). The XPDL version 1.0 was published by the WfMC in 2002. Afterwards it was extended with the goal to be able to represent all concepts of a BPMN diagram in XML. This extension is called XPDL 2.0 and was ratified in 2005.

XPDL in version 2.0 is intended to be used as file format for BPMN. It provides an XML representation for all elements that are part of BPMN. That is why BPMN+ elements and attributes that are also present in BPMN can be transformed to XPDL straightforward. The table below lists the BPMN+ elements and the appropriate XPDL elements.

BPMN+ element	XPDL element
Diagram	Package
Process	WorkflowProcess
Graphical Object	-
Pool	Pool
Pool Set	-
Lane	Lane
Artifact	Artifact
Annotation	Artifact
Group	Artifact
Data Object	DataObject
Connecting Object	-
Sequence Flow	Transition
Message Flow	MessageFlow

Association	Association
Compensation Flow	Association
Flow Object	Activity
Gateway	Route
Events	Event
Activity	Activity
Task	Task
Sub-Process	BlockActivity / ActivitySet

Beside the element attributes, XPDL describes the graphical representation of the elements in a textual way. This is done with the `NodeGraphicsInfo` and `ConnectorGraphicsInfo` elements that can be defined for every graphical object.

In XPDL, the data field construct is intended to express the properties attribute that can be defined for BPMN elements. The type of a property has to be defined by the modeler of the diagram and is not restricted in BPMN. In contrast to this, the types of data fields are restricted to basic types (string, float, integer, date time, boolean, performer), a reference to an already declared type, a reference to an external declared type or an XML schema type declared only for this data field. In BPMN+, declared types cannot be defined that may be referenced by a property. So regarding the mapping to XPDL4Chor, the types of a BPMN+ property have to be restricted to basic, external and schema types. An example for the XPDL representation of a property with the type “string” and a property with an XML schema type is shown in Listing 3.1. The nesting of data field elements is not supported in the XPDL schema. So the child properties of a property with the type “Set” have to be expressed as XML schema type or in the external type.

```
<xpdl:DataField Id="Id2" Name="propertyBasic">
    <xpdl:DataType>
        <xpdl:BasicType Type="STRING"/>
    </xpdl:DataType>
</xpdl:DataField>
<xpdl:DataField Id="Id1" Name="propertySchema">
    <xpdl:DataType>
        <xpdl:SchemaType>
            <schema xmlns="http://www.w3c.org/2001/XMLSchema">
                <element name="example">
                    <complexType>
                        <attribute name="sampleAttribute"
                            type="string"/>
                    </complexType>
                </element>
            </schema>
        </xpdl:SchemaType>
    </xpdl:DataType>
</xpdl:DataField>
```

Listing 3.1: BPMN+ properties with basic and XML schema type in XPDL

Some attributes and elements in XPDL are mandatory, but they are not used in BPMN$^+$ (e.g. MI_Condition attribute of a multi-instance loop). These will be left empty in the case of mandatory attributes. In the case of mandatory elements, there will be used dummy elements with no additional attributes defined.

In addition to the already existing elements, the XPDL schema allows the extension of elements and attributes for vendor specific implementations. This mechanism will be used to add the elements and attributes needed for describing the choreography specific details of a BPMN$^+$ diagram. This extension is introduced as XPDL4Chor and contains the BPMN$^+$ elements and attributes that are not part of BPMN. Unfortunately these extensions are not allowed in every schema construct, so that new task or event types cannot be defined in the construct of the existing ones. To extend the XPDL schema, there have to be identified proper locations for the definition of additional elements and attributes.

Changing the semantics, as done for event handlers, is not problematic in XPDL whereas it is problematic for BPMN. The reason is that according to the XPDL schema, processes or sub-processes can contain activities that are not connected with the sequence flow. The semantic of the elements comes from BPMN and is not described in detail in XPDL. XPDL is more focused on the describing the diagram structure than on describing its semantic. That is why the extension of XPDL with event handlers still conforms to the XPDL specification.

3.2 Mapping BPMN$^+$ Elements to XPDL4Chor

In this section the extension of XPDL to XPDL4Chor is explained and it will be described how BPMN$^+$ elements can be mapped to XPDL4Chor.

A BPMN$^+$ diagram is defined as package in XPDL. Beside the message flows, associations and artifacts are specified in a package and not in a process as in BPMN$^+$. There are some BPMN$^+$ diagram attributes missing in the definition of a package. Thus, a `Language` and `ModificationDate` attribute is added to the `PackageHeader` element and a `QueryLanguage` and `ExpressionLanguage` attribute is added to the `RedefinableHeader` element. In addition to this, the introduced BPMN$^+$ diagram attributes PoolSet, TargetNamespace and GroundingFile are added to an XPDL4Chor package.

In XPDL, a workflow process holds the graphical elements that are contained within a pool or pool set (except artifacts and associations). In BPMN$^+$, properties of the type “Set” with the Correlation attribute set to “True” are used to define the correlation sets of a process. The type “Set” indicates the nesting of properties, whereas the child properties denote the correlation properties of the correlation set. As mentioned above only basic, external or XML schema types can be mapped to XPDL4Chor. Moreover, data field elements, which represent the BPMN$^+$ properties, can not be nested. Thus, this kind of a BPMN$^+$ property has to be treated in a special way. In Listing 3.2 a sample data field can be seen that represents a correlation set property. The schema type of this data field specifies elements that represent the child properties of the correlation set. Like in BPMN$^+$, the types of the correlation set properties do not need to be specified.

Since a pool represents a single participant in BPMN$^+$, it has appropriate attributes that must be added to a pool in XPDL4Chor. In addition to this, a pool set has to be introduced. It looks nearly similar to the specification of a pool in XPDL4Chor, but the attributes concerning the participant are not present. A sample pool set in XPDL4Chor can be seen in Listing 3.3. For both, pools and pool sets, the `imports` element is introduced that provides information about external XML schema or WSDL files.

```
<xpdl:DataField Id="id1" Name="CorrelationSet"
    Correlation="true">
    <xpdl:DataType>
        <xpdl:SchemaType>
            <schema xmlns="http://www.w3.org/2001/XMLSchema">
                <element name="correlation_property1"/>
                <element name="correlation_property2"/>
            </schema>
        </xpdl:SchemaType>
    </xpdl:DataType>
</xpdl:DataField>
```

Listing 3.2: BPMN+ property for correlation in XPDL4Chor

A lane can be directly mapped to XPDL4Chor, but in addition to the pool element, a pool set element has to be introduced in its schema definition.is build

```
<chor:PoolSet Id="PoolSet1" Name="Suppliert"
    BoundaryVisible="true" Orientation="VERTICAL"
    ProcessLanguage="urn:HPI_IAAS:choreography:profile:2006/12"
    TargetNamespace="http://example.com/supplier"
    Prefix="sup" Process="process3">
    <xpdl:Lanes>
        <xpdl:Lane Id="Lane_Pool1" Name="supplierLane"
            ParentPool="Pool1" />
    </xpdl:Lanes>
    <chor:Imports>
        <chor:Import Prefix="sup"
            Namespace="http://example.com/wsdls/supplier"
            ImportType="http://schemas.xmlsoap.org/wsdl" />
    </chor:Imports>
</chor:PoolSet>
```

Listing 3.3: A pool set in XPDL4Chor

Artifacts can be associated with a pool or a pool set in BPMN+. In XPDL, this cannot be done, because artifacts are defined on the package level and not for a special pool or pool set. That is why additional attributes have to be introduced in the XPDL4Chor schema to express this relationship. Despite this, artifacts of the type annotation and group can be mapped directly to XPDL4Chor. Data object artifacts are mapped to XPDL4Chor using the `DataObject` element that can be defined for an artifact. The special data object types for variables, participant references and participant sets can be distinguished with the additional `VariableDataObject`, `ParticipantReferenceDataObject` and `ParticipantSetDataObject` elements. These elements are introduced for a `DataObject` element in XPDL4Chor.

`VariableDataObject` elements have a `VariableTypeValue` attribute to define the type of the variable. The value of this attribute is built up of the DefinitionTypePrefix and DefinitionTypeValue attributes of a variable data object in BPMN+. The prefix of the variable type must be associated with a namespace in an `Import` element of the pool or

pool set the data object is located in. A sample variable data object in XPDL4Chor is given in Listing 3.4 and the appropriate import is defined in the pool set of Listing 3.3.

Although a DataField element already provides a child element to initialize the variable with a value, it cannot be reused for the mapping from BPMN+. The initialization in BPMN+ is much more complex than the expression provided for this purpose in XPDL. That is why a FromSpec element has to be introduced for a VariableDataObject element, the attributes of the BPMN+ FromSpec type can be mapped to.

```
<xpdl:Artifact Id="Artifact7" Name="variable"
    ArtifactType="DataObject" chor:Process="resource0_process"
    chor:Pool="resource0">
    <xpdl:DataObject Id="Artifact7_DataObject"
        RequiredForStart="true" ProducedAtCompletion="true">
        <chor:VariableDataObject Type="Standard"
            VariableType="MessageType"
            VariableTypeValue="sup:request" />
    </xpdl:DataObject>
</xpdl:Artifact>
```

Listing 3.4: Variable data object in XPDL4chor

Mapping the connecting objects from BPMN+ to XPDL4Chor is as simple as for most of the other elements and there are no additional attributes necessary. The only difference is that a sequence flow is called transition in XPDL4Chor.

Flow objects are represented as activities in XPDL and they are defined in activity sets or workflow processes. Like artifacts, flow objects do not have a direct relation to the pool or pool set they belong to. However, in contrast to artifacts this can be determined from the process or activity set they are defined in. In addition to this, an attribute has to be introduced that identifies the lane that the flow object is located in.

A gateway maps to an XPDL Route element, whereas the GatewayType attribute distinguishes parallel, inclusive and exclusive gateways. The Split element can be defined for a TransitionRestriction element of an activity. It is used to distinguish event-based and data-based exclusive gateways. The gates of the gateways are defined as normal transitions that may have conditions defined in the case of data-based exclusive or inclusive decision gateways. To specify the evaluation order for the outgoing transitions of a data-based exclusive decision gateway, the TransitionRestriction element must contain TransitionReference elements. These are referencing outgoing transitions of the gateway. The order of these references in the TransitionRestriction element defines the evaluation order. An example for a data-based exclusive decision and an exclusive merge gateway can be seen in Listing 3.5.

BPMN+ events can be mapped onto XPDL4Chor Event elements defined for an activity. The elements StartEvent, EndEvent and IntermediateEvent are used to distinguish the different event types. Like in BPMN+, the trigger for intermediate termination events has to be introduced in XPDL4Chor. XPDL does not provide the ability to extend the already existing trigger types. That is why the termination event is introduced using the new attribute IsTermination. If this attribute is set to “true”, the value of the Trigger attribute is irrelevant.

```
<xpdl:Activity Id="process1_activity1">
    <xpdl:Route GatewayType="XOR" MarkerVisible="true" />
    <xpdl:TransitionRestrictions>
        <xpdl:TransitionRestriction>
            <xpdl:Split Type="XOR" >
                <xpdl:TransitionRefs>
                    <xpdl:TransitionRef
                            Id="process1_transition2"/>
                    <xpdl:TransitionRef
                        Id="process1_transition3"/>
                </xpdl:TransitionRefs>
            </xpdl:Split>
        </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
</xpdl:Activity>

<xpdl:Activity Id="process1_activity7">
    <xpdl:Route GatewayType="XOR" MarkerVisible="true" />
</xpdl:Activity>
```

Listing 3.5: Event-based decision and merge gateway in XPDL4Chor

In BPMN+, timer events are extended with a `TimeLanguage` attribute. This has to be done in XPDL4Chor for the `TriggerTimer` element. The `RepeatEvery` attribute is added as new element of the type `ExpressionType`. The additional attributes introduced for an event triggered by a message in BPMN+ must be added in XPDL4Chor for a `TriggerResultMessage` element.

An activity in the terms of BPMN+ maps to an XPDL4Chor activity. Tasks are defined in the `Implementation` element of an activity. Unfortunately, the XPDL schema does not allow an extension of the already existing task types. Thus, a new element called `Task` has to be introduced that provides child elements holding the attributes of the new task types from BPMN+. These child elements are called `TaskAssign`, `TaskEmpty` and `TaskValidate`. XPDL does not allow non-typed tasks in its schema so an element `TaskNone` has to be introduced as well. Listing 3.6 depicts an example for an empty task in XPDL4Chor.

```
<xpdl:Activity Id="process3_activity4" Name="empty"
    <xpdl:Extensions />
    <chor:Task>
        <chor:TaskEmpty />
    </chor:Task>
</xpdl:Activity>
```

Listing 3.6: Empty task in XPDL4Chor

A sub-process in BPMN+ maps to an XPDL4Chor `BlockActivity` element. The graphical elements of a block activity (activities and transitions) are defined in an `ActivitySet` element. This element is located in the `WorkflowProcess` element of the process that the sub-process belongs to. The BPMN+ sub-process types scope and handler are not distinguished in XPDL. Thus, special elements called `Scope` and `Handler` are added for a `BlockActivity` element that hold the appropriate attributes.

The attributes for looping activities are located in the child elements of an activity called `LoopStandard` and `LoopMultipleInstance`. The `MI_Condition` attribute is mandatory in XPDL4Chor, but omitted in BPMN^{+}. That is why it will be left empty during the mapping. The `LoopCondition` attribute in XPDL4Chor is just a string value whereas in BPMN^{+} it is an expression. Thus, a `LoopConditionLanguage` attribute is added to define the expression language. Additional attributes and elements for a multi-instance loop (`StartCounterValue`, `FinalCounterValue`, `CompletionCondition` and `SuccessfulBranchesOnly`) are introduced in XPDL4Chor as well.

Supporting types like Import, Copy, FromSpec, ToSpec, which were introduced in BPMN^{+}, have to be added in XPDL4Chor, too. The expression type like in BPMN^{+} is already present in XPDL, so it does not need to be changed. In XPDL, the expression language is called `ScriptGrammar`.

4 BPMN+ to BPEL4Chor

Now that the input format for the transformation is defined, the actual transformation from BPMN+ to BPEL4Chor can be developed. There is already some effort to transform BPMN to BPEL: Mendling et al. has sketched four different transformation strategies from BPMN to BPEL in [MLZ05]: element-preservation, element-minimization, structure-identification and structure-maximization. The *element-preservation* strategy maps an acyclic graph to a BPEL `flow` construct. The graph elements are mapped to the appropriate BPEL activities and added to the `flow`. Gateways are mapped to empty activities. The sequence flows between the elements are mapped to control links in the `flow` element. The *element-minimization* strategy removes the empty activities that were generated for the gateways in the element-preservation strategy. Instead, transition conditions are added to the control links and join conditions to the subsequent activities. The *structure-identification* strategy identifies structured activities in the process graph and applies structural reduction rules to them. The *structure-maximization* strategy applies the reduction rules of the structure-identification strategy as often as possible to determine a maximum of structured activities.

In the BPMN specification [BPMN06], a very informal transformation is described following the structure-identification strategy. The transformation of White ([Whi05]) considers the element attributes. It suggests either to follow the element-minimization or the structure-identification strategy to map the sequence flow. The most formal transformation algorithm is given by Ouyang et al. in [ODtHvdA07]. This algorithm follows the structure-maximization strategy and the remaining components are transformed using the element-minimization strategy.

In this section the transformation of Ouyang et al. will be adapted to BPMN+. This extension results in a transformation that can be used for the sequence flow in the processes and sub-processes of a BPMN+ choreography. The generation of the topology from the diagram is put on top of this transformation.

4.1 BPMN to BPEL

The transformation of Ouyang et al. defined in [ODtHvdA07] covers the fundamental control flow elements of BPMN. These are:

- non-triggered start events
- intermediate events (message and timer)
- tasks
- parallel gateways (fork and join)
- exclusive gateways (data-based decision, event-based decision, merge)
- sequence flows

To simplify the transformation, a diagram consisting of these elements is restricted to a well-formed core BPD. These restrictions concern the sequence flow connected with the BPMN elements.

The graph structure of a BPMN diagram is transformed into the basic block structures of BPEL by identifying "patterns" in the diagram that can be mapped onto BPEL blocks.

During the transformation, components are identified that match these patterns. Such a component is folded to a single task that references the BPEL code it represents. This folded task replaces the component within the sequence flow. This is done until no pattern is left.

There are three different types of patterns: *Well-structured* patterns can be mapped to BPEL `sequence`, `flow`, `if`, `pick`, `while` and `repeatUntil` activities. Components matching *quasi-structured* patterns can be converted into components that contain well-structured patterns. More complex BPEL `flows` are mapped using the *generalized flow* pattern.

In addition to the pattern-based translations, there are two approaches defined for components that do not match any of the patterns above: The *control link-based translation* is applied to acyclic components without event-based decision gateways. As a last resort, the *event-action rule-based translation* supports the translation of components that contain arbitrary cycles.

The transformation described in this thesis, is based on the transformation depicted above. In contrast to this transformation, there is no claim to transform any construct that can be modeled in BPMN. For example, the transformation of arbitrary cycles to BPEL is a problem that cannot be solved in a simple way. This can be seen in the event-action rule based translation that exclusively uses event handlers to express the control flow elements. This results in BPEL code that is very convoluted. That is why only the pattern based transformation (well-structured, quasi-structured and generalized flow) and the control link-based translation will be used. In the next section the extension of this transformation will be described to support BPMN$^+$ diagrams.

4.2 Generating the BPEL4Chor Processes

The adapted Ouyang et al. transformation will be used for the sequence flow located in pools, pool sets and sub-processes. The transformation described in this section does not transform the whole diagram. That is why the term “BPD” used in the transformation gets replaced with the term “process”.

To define the patterns in a formal way a definition of a process is given in Definition 1. This is the definition of a BPD from the Ouyang et al. transformation extended with BPMN$^+$ specific elements:

- start events: message and timer start events
- intermediate events: error and compensation intermediate events
- tasks: other task types in addition to the receive task
- gateways: inclusive decision and inclusive merge gateways
- sub-processes as scopes or fault handlers
- attached error events

Only fault handlers occur in the definition, because other handlers are not necessary for identifying the patterns. Termination and compensation handlers are mapped directly when the task or scope is mapped they are connected with. Event handlers do not occur because they are not part of the actual sequence flow. They are mapped before the transformation of the sequence flow starts. A more detailed description of the mapping of handlers is given in section see 4.2.4.

Definition 1 (Process): A process is a tuple $\mathcal{PS} = (\mathcal{O}, \mathcal{F}, \mathsf{Cond}, \mathsf{Att})$ where:

- $\mathcal{O}$ is a set of objects which is divided into disjoint sets of tasks $\mathcal{T}$, events $\mathcal{E}$, gateways $\mathcal{G}$ and sub-processes $\mathcal{P}$,
- $\mathcal{T}$ is divided into disjoint sets of service tasks $\mathcal{T}^I$, receive tasks $\mathcal{T}^R$, send tasks $\mathcal{T}^P$, assign tasks $\mathcal{T}^A$, empty tasks $\mathcal{T}^E$, validate tasks $\mathcal{T}^V$ and non-typed tasks $\mathcal{T}^O$, whereas $(\mathcal{T}^I \cup \mathcal{T}^R \cup \mathcal{T}^P \cup \mathcal{T}^A \cup \mathcal{T}^E \cup \mathcal{T}^V \cup \mathcal{T}^O) \subseteq \mathcal{T}$
- $\mathcal{E}$ is divided into disjoint sets of start events $\mathcal{E}^S$, intermediate events $\mathcal{E}^I$ and end events $\mathcal{E}^E$,
- $\mathcal{E}^S$ is divided into disjoint sets of non-triggered start events $\mathcal{E}^S_N$, message start events $\mathcal{E}^S_M$ and timer start events $\mathcal{E}^S_T$,
- $\mathcal{E}^I$ is divided into disjoint sets of message intermediate events $\mathcal{E}^I_M$, timer intermediate events $\mathcal{E}^I_T$, error intermediate events $\mathcal{E}^I_E$ and compensation intermediate events $\mathcal{E}^I_C$,
- $\mathcal{G}$ is divided into disjoint sets of parallel fork gateways $\mathcal{G}^F$, parallel join gateways $\mathcal{G}^J$, data-based exclusive decision gateways $\mathcal{G}^D$, event-based exclusive decision gateways $\mathcal{G}^V$, exclusive merge gateways $\mathcal{G}^M$, inclusive decision gateways $\mathcal{G}^{ID}$ and inclusive merge gateways $\mathcal{G}^{IM}$,
- $\mathcal{P}$ is divided into disjoint sets of scopes $\mathcal{P}^S$ and fault handlers $\mathcal{P}^F$
- $\mathcal{F} \subseteq \mathcal{O} \times \mathcal{O}$ is the control flow relation, i.e., a set of sequence flows connecting objects,
- $\mathsf{Cond} : \mathcal{F} \cap ((\mathcal{G}^D \cup \mathcal{G}^{ID}) \times \mathcal{O}) \to \mathcal{B}$ is a function mapping sequence flows emanating from data-based exclusive decision gateways or inclusive decision gateways to the set of all possible conditions ($\mathcal{B}$),
- $\mathsf{Att} : \mathcal{E}^I_E \nrightarrow (\mathcal{T} \cup \mathcal{P}^S)$ is the attachment function, i.e., intermediate error events can be attached to tasks and scopes.

The relation $\mathcal{F}$ defines a directed graph with nodes $\mathcal{O}$ and arcs. For any $x \in \mathcal{O}$, input nodes of x are given by $\mathsf{in}(x) = \{y \in \mathcal{O} \mid y\mathcal{F}x\}$ and output nodes of x are given by $\mathsf{out}(x) = \{y \in \mathcal{O} \mid x\mathcal{F}y\}$. The transformation only accepts processes that comply with some restrictions to the control flow relation. These restrictions are captured in Definition 2. This is the definition of a well-formed BPD from the Ouyang et al. transformation extended with the BPMN+ specific details regarding attached events.

Definition 2 (Well-formed process): A process is well-formed iff relation $\mathcal{F}$ satisfies the following requirements:

- $\forall s \in \mathcal{E}^S$, $\mathsf{in}(s) = \emptyset \wedge |\mathsf{out}(s)| = 1$, i.e., start events have an indegree of zero and an outdegree of one,
- $\forall e \in \mathcal{E}^E$, $\mathsf{out}(e) = \emptyset \wedge |\mathsf{in}(e)| = 1$, i.e., end events have an outdegree of zero and an indegree of one,
- $\forall x \in \mathcal{T} \cup \mathcal{P}$, $|\mathsf{in}(x)| = |\mathsf{out}(x)| = 1$, i.e., tasks and sub-processes have an indegree of one and an outdegree of one,
- $\forall g \in \mathcal{G}^F \cup \mathcal{G}^D \cup \mathcal{G}^V \cup \mathcal{G}^{ID}$, $|\mathsf{in}(g)| = 1$ and $|\mathsf{out}(g)| > 1$, i.e., fork and decision gateways have an indegree of one and an outdegree of more than one,
- $\forall g \in \mathcal{G}^J \cup \mathcal{G}^M \cup \mathcal{G}^{IM}$, $|\mathsf{out}(g)| = 1$ and $|\mathsf{in}(g)| > 1$, i.e., join and merge gateways have an outdegree of one and an indegree of more than one,

– $\forall g \in \mathcal{G}^V$, $\mathsf{out}(g) \subseteq \mathcal{E}^I_M \cup \mathcal{E}^I_T \cup \mathcal{T}^R$, i.e., event-based exclusive decision gateways must be followed by receive tasks, message intermediate or timer intermediate events,
– $\forall g \in \mathcal{G}^D$, $\exists$ an order $<_g$ which is a strict total order over the set of flows $\{g\} \times \mathsf{out}(g)$ and for $x \in \mathsf{out}(g)$ such that $\neg\exists_{f\in\{g\}\times\mathsf{out}(g)}((g,x) <_g f)$, (g,x) is the default flow among all the outgoing flows from g,
– $\forall x \in \mathcal{O}$, $\exists s \in \mathcal{E}^S \cup dom(\mathsf{Att})$, $\exists e \in \mathcal{E}^E$, $s\mathcal{F}^*x \wedge x\mathcal{F}^*e$, i.e., every flow object is on a path from a start event or attached error intermediate event to an end event,
– $\forall x \in \mathcal{E}^I, |\mathsf{out}(x)| = 1$ and $|\mathsf{in}(x)| = \begin{cases} 0 & \text{if } x \in dom(\mathsf{Att}) \\ 1 & \text{otherwise} \end{cases}$,
i.e., attached intermediate events have an indegree of zero,
– $\forall (x,y) \in \mathcal{F} : x \in dom(\mathsf{Att}) \Leftrightarrow y \in \mathcal{P}^F$, i.e., each attached error intermediate event is connected with a fault handler and each fault handler is connected with an attached error intermediate event.

Att is a partial function. The domain of Att, expressed with dom(Att), are all intermediate error events the function Att is defined for. This means that the domain of Att represents all attached intermediate error events.

The transformation can only transform processes that correspond to the meta-model described in chapter 2.2. Moreover, the restrictions given in Definition 1 and Definition 2 must be followed. Also, as stated above, arbitrary cycles in the process control flow cannot be transformed. In addition to this, it is assumed that the graph is sound and safe. Sound means that the graph must have no deadlocks. A safe graph has no multi-instances of the same activity that execute concurrently (except for multi-instance activities). An example for a process that is not sound is shown in Figure 4.1. A parallel fork gateway followed by an exclusive merge gateway results in multiple tokens where each of them instantiates the task C. More information about the checking of soundness and safeness can be found in section 4.2.6.

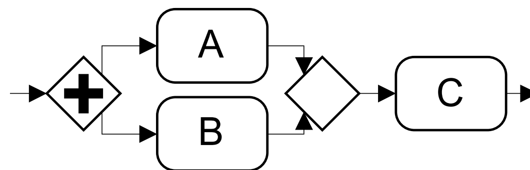


Figure 4.1: Non-safe sequence flow

4.2.1 Combine Multiple Start and End Events

The original transformation expects that the sequence flow has one start and one end event. That is why multiple start and end events have to be combined before the sequence flow can be transformed.

Multiple message start events reflect the scenario where a process can be instantiated through the receipt of different messages. Multiple non-triggered start events are not allowed. Timer start events only occur in timer event handlers. Since event handlers are not allowed to have multiple start events, the transformation does not have to consider multiple timer start events. That is why only multiple message triggered start events are combined.

The combination of two or more start events depends on the gateway that merges the sequence flow emanating from these start events. This is the first gateway between the start and the end event that can be reached from other start events. If this gateway is a parallel join gateway, the start events can be combined using a parallel split gateway. This will later match to a (quasi) flow pattern in the actual sequence flow transformation. If the gateway is an exclusive merge gateway, the start events can be combined using an event-based decision gateway. This results in a (quasi) pick pattern. Inclusive gateways are not allowed for merging the sequence flow of multiple start events. The definition of inclusive merge gateways in the BPMN specification says it waits for all tokens that have been produced upstream. From a point of view that is restricted to the process context, it is not known at any time whether a token will be produced from a message triggered start event. So the inclusive merge cannot determine when the merge should take place.

There must be at least one gateway merging the sequence flow from the start events. If not, there will be independent process flows that are not connected with each other. Such a diagram cannot be transformed to BPEL4Chor. It is not clear when the process will terminate, because although one process flow has finished the other start events might still be triggered.

Using event-based decision gateways for the combination of start events, the semantic of the model gets modified. The exclusive merge without an event-based decision split allows that the flow will continue each time a token arrives from one of the incoming sequence flows (multi-merge workflow pattern, [RtHvdAM06]). Structured with an event-based decision gateway, there can only arrive one token at any time. As stated in [WvdA+02] BPEL, and so BPEL4Chor, do not support the multi-merge workflow pattern. That is why it is assumed that this semantic change can be ignored regarding the transformation to BPEL4Chor.

The combination of start events is done as follows: The gateway combining the start events is inserted before the start events such that every branch leads to one start event. Since the start events are now located within the sequence flow, they turn into intermediate events. There is not necessarily one gateway merging the sequence flow of all start events. More likely is that there are different gateways, each merging the sequence flow of a subset of start events. In this case each subset has to be combined separately.

There is a scenario regarding multiple message start events that cannot be transformed with this approach, although the combining gateways are not inclusive. For example, the illustration on the left side in Figure 4.2 shows three start events. The outgoing sequence flows of two start events are merged in a parallel join gateway. The resulting sequence flow is merged again with the sequence flow of the third start event using an exclusive merge gateway. The result of the combination of these start events is illustrated on the right sight of the figure. The event-based decision gateway leads to a parallel gateway. This is not allowed in BPMN$^+$, where each event-based decision gateway has to be followed by intermediate events or receive tasks.

The combination of multiple end events can be done using an inclusive merge gateway. An inclusive merge gateway normally waits on each of its incoming branches for a token that indicates the completion of this branch. If at some point in time it can be determined that no token will ever arrive along an incoming branch, the inclusive merge will not wait for a token along that branch anymore. With this semantic each sequence flow can be merged that was branched before by any gateway type.

The combination of multiple start and end events does not lead to a diagram that can be transformed to BPEL4Chor in any case. If the diagram contains constructs that are not covered by the defined transformation patterns (e.g. multiple token flow, arbitrary

cycles), these constructs are still present after the combination of the multiple start and end events.

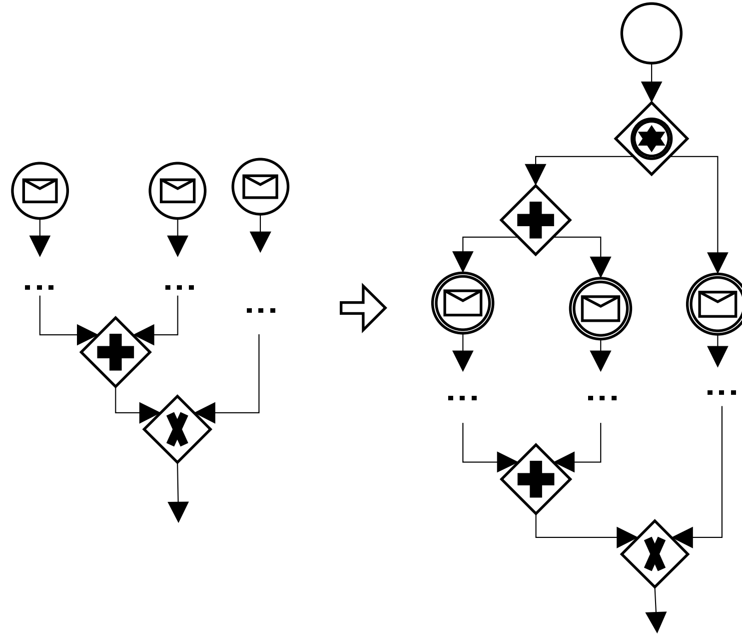


Figure 4.2: Invalid combination of multiple start evens

4.2.2 Extended Patterns

The transformation tries to transform the graph structure of BPMN+ into the block structure of BPEL4Chor. For this purpose it identifies components that match defined patterns. These components can be mapped onto BPEL4Chor blocks. Ouyang et al. already defines a set of patterns. These patterns will be extended in the next sections to support all elements that are allowed in BPMN+.

A component is a subset of a process that has one entry and one exit point. The formalization of a component is given in Definition 3. The definition uses the auxiliary function elt over a domain of singletons as defined in [ODtHvdA07], i.e., if $X = \{x\}$, then $\mathsf{elt}(X) = x$. In addition to the original component definition from Ouyang et al., the definition below considers the function Att that was already defined in Definition 1.

Definition 3 (Component): Let $\mathcal{PS} = (\mathcal{O}, \mathcal{F}, \mathsf{Cond}, \mathsf{Att})$ be a well-formed process. A subset of $\mathcal{PS}$, as given by $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c, \mathsf{Att}_c)$ is a component iff:

- $\mathcal{O}_c \subseteq \mathcal{O} \setminus (\mathcal{E}^S \cup \mathcal{E}^E)$, i.e., a component does not contain any start or end event,
- $|(\bigcup_{x \in \mathcal{O}_c} \mathsf{in}(x)) \setminus \mathcal{O}_c| = 1$, i.e., there is a single entry point into the component, which can be denoted as $\mathsf{entry}(\mathcal{C}) = \mathsf{elt}((\bigcup_{x \in \mathcal{O}_c} \mathsf{in}(x)) \setminus \mathcal{O}_c)$,
- $|(\bigcup_{x \in \mathcal{O}_c} \mathsf{out}(x)) \setminus \mathcal{O}_c| = 1$, i.e., there is a single exit point out of the component, which can be denoted as $\mathsf{exit}(\mathcal{C}) = \mathsf{elt}((\bigcup_{x \in \mathcal{O}_c} \mathsf{out}(x)) \setminus \mathcal{O}_c)$,
- there exists a unique source object $i_c \in \mathcal{O}_c$ and a unique sink object $o_c \in \mathcal{O}_c$ and $i_c \neq o_c$ such that $\mathsf{entry}(\mathcal{C}) \in \mathsf{in}(i_c)$ and $\mathsf{exit}(\mathcal{C}) \in \mathsf{out}(o_c)$,
- $\mathcal{F}_c = \mathcal{F} \cap (\mathcal{O}_c \times \mathcal{O}_c)$
- $\mathsf{Cond}_c = \mathsf{Cond}\,[\mathcal{F}_c]$, i.e., the Cond function where the domain is restricted to $\mathcal{F}_c$

– $\mathsf{Att}_c = \mathsf{Att}[\mathcal{E}^I_{Ec}]$, i.e., the Att function where the domain is restricted to $\mathcal{E}^I_{Ec}$.

To extend the original transformation with the BPMN+ specific elements, the patterns the transformation is based on have to be extended. Since inclusive gateways are not part of the original transformation, the existing patterns have to be extended with this gateway type to allow multiple end events.

Definition 4 shows the extended well-structured patterns. The sequence, flow, if, pick, while, repeat and repeat-while patterns were taken from the original transformation. The special-flow- and fault-handler-patterns are introduced for the transformation from BPMN+ to BPEL4Chor. Inclusive gateways can be used instead of the other merging gateway types in all patterns. In addition to tasks and intermediate events, scopes can be objects within the sequence flow.

The special-flow pattern is introduced to support the usage of inclusive split gateways in BPMN+ diagrams. It is similar to the flow-pattern, but the source object is an inclusive split gateway instead of a parallel fork gateway.

In BPEL4Chor the process execution continues after a faulted activity if the fault was caught. That is why in BPMN+ every fault handler must lead to an exclusive or inclusive merge gateway to merge the exception flow with the normal sequence flow after the faulted activity. This case is reflected in the fault-handler pattern. The source object is the task or sub-process that error events are attached to. The sink object is the gateway that merges the exception flow with the normal sequence flow.

Definition 4 (Well-structured patterns): Let $\mathcal{PS} = (\mathcal{O}, \mathcal{F}, \mathsf{Cond}, \mathsf{Att})$ be a well-formed process and $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c, \mathsf{Att}_c)$ be a component of $\mathcal{PS}$. i_c is the source object of $\mathcal{C}$ and o_c is the sink object of $\mathcal{C}$. The following components are identified as well-structured patterns:

(a) $\mathcal{C}$ exhibits a SEQUENCE-pattern iff $\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{E}^I \cup \mathcal{P}^S$, i.e., $\forall x \in \mathcal{O}_c$, $|\mathsf{in}(x)| = |\mathsf{out}(x)| = 1$, $\mathsf{entry}(\mathcal{C}) \notin \mathcal{G}^V$ and $dom(\mathsf{Att}_c) = \emptyset$. $\mathcal{C}$ is a maximal SEQUENCE-pattern iff $\mathcal{C}$ is a SEQUENCE-pattern and there is no other SEQUENCE-pattern $\mathcal{C}'$ such that $\mathcal{O}_c \subset \mathcal{O}'_c$ where $\mathcal{O}'_c$ is the set of objects in $\mathcal{C}'$

(b) $\mathcal{C}$ is identified as a FLOW-pattern iff:
- $i_c \in \mathcal{G}^F \wedge o_c \in (\mathcal{G}^J \cup \mathcal{G}^{IM})$,
- $\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{E}^I \cup \mathcal{P}^S \cup \{i_c, o_c\}$,
- $\forall x \in \mathcal{O}_c \setminus \{i_c, o_c\}, \mathsf{in}(x) = \{i_c\} \wedge \mathsf{out}(x) = \{o_c\}$
- $dom(\mathsf{Att}_c) = \emptyset$.

(c) $\mathcal{C}$ is identified as a SPECIAL-FLOW-pattern iff:
- $i_c \in \mathcal{G}^{ID} \wedge o_c \in \mathcal{G}^{IM}$,
- $\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{E}^I \cup \mathcal{P}^S \cup \{i_c, o_c\}$,
- $\forall x \in \mathcal{O}_c \setminus \{i_c, o_c\}, \mathsf{in}(x) = \{i_c\} \wedge \mathsf{out}(x) = \{o_c\}$
- $dom(\mathsf{Att}_c) = \emptyset$

(d) $\mathcal{C}$ is identified as IF-pattern iff:
- $i_c \in \mathcal{G}^D \wedge o_c \in (\mathcal{G}^M \cup \mathcal{G}^{IM})$,
- $\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{E}^I \cup \mathcal{P}^S \cup \{i_c, o_c\}$,
- $\forall x \in \mathcal{O}_c \setminus \{i_c, o_c\}, \mathsf{in}(x) = \{i_c\} \wedge \mathsf{out}(x) = \{o_c\}$
- $dom(\mathsf{Att}_c) = \emptyset$.

(e) $\mathcal{C}$ is identified as PICK-pattern iff:

- $i_c \in \mathcal{G}^V \wedge o_c \in (\mathcal{G}^M \cup \mathcal{G}^{IM})$,
- $\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{P}^S \cup \mathcal{E}^I_M \cup \mathcal{E}^I_T \cup \{i_c, o_c\}$,
- $\forall x \in \mathcal{O}_c \setminus (\{i_c, o_c\} \cup \mathsf{out}(i_c)), \mathsf{in}(x) \subset \mathsf{out}(i_c) \wedge \mathsf{out}(x) = \{o_c\}$
- $dom(\mathsf{Att}_c) = \emptyset$.

(f) $\mathcal{C}$ is identified as WHILE-pattern iff:
- $i_c \in (\mathcal{G}^M \cup \mathcal{G}^{IM}) \wedge o_c \in \mathcal{G}^D \wedge x \in \mathcal{T} \cup \mathcal{P}^S \cup \mathcal{E}^I$,
- $\mathcal{O}_c = \{i_c, o_c, x\}$,
- $\mathcal{F}_c = \{(i_c, o_c), (o_c, x), (x, i_c)\}$
- $dom(\mathsf{Att}_c) = \emptyset$.

(g) $\mathcal{C}$ is identified as REPEAT-pattern iff:
- $i_c \in (\mathcal{G}^M \cup \mathcal{G}^{IM}) \wedge o_c \in \mathcal{G}^D \wedge x \in \mathcal{T} \cup \mathcal{P}^S \cup \mathcal{E}^I$,
- $\mathcal{O}_c = \{i_c, o_c, x\}$,
- $\mathcal{F}_c = \{(i_c, x), (x, o_c), (o_c, i_c)\}$
- $dom(\mathsf{Att}_c) = \emptyset$.

(h) $\mathcal{C}$ is identified as REPEAT-WHILE-pattern iff:
- $i_c \in (\mathcal{G}^M \cup \mathcal{G}^{IM}) \wedge o_c \in \mathcal{G}^D$,
- $x_1 \in (\mathcal{T} \cup \mathcal{P}^S \cup \mathcal{E}^I) \setminus (\mathcal{T}^I \cup \mathcal{T}^R \cup \mathcal{T}^P \cup \mathcal{E}^I_M) \wedge x_2 \in \mathcal{T} \cup \mathcal{P}^S \cup \mathcal{E}^I$ $\wedge x_1 \neq x_2$
- $\mathcal{O}_c = \{i_c, o_c, x_1, x_2\}$
- $\mathcal{F}_c = \{(i_c, x_1), (x_1, o_c), (o_c, x_2), (x_2, i_c)\}$
- $dom(\mathsf{Att}_c) = \emptyset$.

(i) $\mathcal{C}$ is identified as FAULT-HANDLER-pattern iff:
- $i_c \in (\mathcal{T} \cup \mathcal{P}^S) \wedge o_c \in (\mathcal{G}^M \cup \mathcal{G}^{IM})$
- $\mathcal{W}_c = \{x \in \mathcal{E}^I_E | (x, i_c) \in \mathsf{Att}_c\} \neq \emptyset$, i.e., error events attached to the source object
- $Y = \{y \in \mathcal{P}^F | \exists x \in \mathcal{W}_c : (x, y) \in \mathcal{F}\}$, i.e. fault handlers connected to the attached error events
- $\mathcal{O}_c = \{i_c, o_c\} \cup \mathcal{W}_c \cup Y$
- $\mathsf{in}(o_c) = Y \cup \{i_c\}$

The table below shows examples for the mapping of a special-flow-, repeat-while- and fault-handler-pattern to BPEL4Chor. The mapping of a repeat-while-pattern (h) combines the repeat loop of task x_1 with the while loop of task x_2. It can be seen that the resulting BPEL4Chor code uses the mapping of the task x_1 twice (Mapping(x_1) denotes the BPEL4Chor code for the mapping of task x_1). This results in two BPEL4Chor activities having the same attributes. In the case of communicating activities this leads to two communicating activities having the same name. BPEL4Chor requires unique naming for communicating activities. That is why x_1 in a repeat-while-pattern cannot be a service task, receive task, send task or intermediate message event.

The mapping of the fault-handler-pattern depends on the error code attribute that is defined for each attached error event. There can be one error event that does not have an error code attribute defined. This event is mapped to a `catchAll` element instead of a `catch` element. The sequence flow within the fault handlers is mapped again with the extended sequence flow transformation explained in this section.

In BPMN$^+$, each task can be attached with error intermediate events. However, in BPEL4Chor not every activity can define a fault handler. For these tasks an additional `scope` element is generated that contains the mapping of the task and its fault handlers.

	BPMN$^+$	BPEL4Chor

(c)		<flow>
		  <links>
		    <link name=”l1”/>
		    ...
		    <link name=”ln”/>
		  </links>
		  <empty>
		    <sources>
		      <source linkName=”l1”>
		          <transitionCondition>
		                c1
		          </transitionCondition>
		      </source>
		      ...
		      <source linkName=”ln”>
		          <transitionCondition>
		                cn
		          </transitionCondition>
		      </source>
		    </sources>
		  </empty>
		  Mapping(x1) -- target of l1
		  ...
		  Mapping(xn) -- target of ln
		</flow>

(h)		<sequence>
		  Mapping(x1)
		  <while>
		    <condition>c1</condition>
		    <sequence>
		      Mapping(x2)
		      Mapping(x1)
		    </sequence>
		  </while>
		</sequence>

(i)		<invoke>
		  <catch faultName=“fault1“>
		    ...
		  </catch>
		  <catchAll>
		    ...
		  </catchAll>
		</invoke>

Components that match quasi-structured patterns can be converted to components that contain well-structured patterns. Definition 5 shows the formalization of the quasi-struc-

tures patterns. It also specifies how to refine them to construct a well-structured pattern within them. The quasi-flow-, quasi-if and quasi-pick patterns were taken from the original transformation. The quasi-special-flow and quasi-fault-handler pattern are introduced for the transformation from BPMN+ to BPEL4Chor. Like for the well-structured patterns, the inclusive merge gateway is added as possible sink object for all patterns. In addition to the tasks and events, scopes are allowed within the sequence flow.

In contrast to the exclusive data-based decision gateway, the inclusive decision gateway does not define an evaluation order that has to be preserved during the refinement. That is why it can be split for the refinement of the quasi-special-flow pattern. However, the sequence flow generated during the refinement has to define a condition. This condition is necessary, because inclusive decision gateways can only be connected with conditional sequence flows. In this case a sequence flow condition will be used that always evaluates to true.

The definition uses the function $\mathsf{succ}(x)$, which provides the object that follows x if x has an outdegree of one.

Definition 5 (Quasi-structured patterns): Let $\mathcal{PS} = (\mathcal{O}, \mathcal{F}, \mathsf{Cond}, \mathsf{Att})$ be a well-formed process and $\mathcal{TEP} = \mathcal{T} \cup \mathcal{E} \cup \mathcal{P}^S$. Five types of quasi-structured patterns may be identified as follows:

(a) Let $x \in \mathcal{G}^F$, $y \in (\mathcal{G}^J \cup \mathcal{G}^{IM})$, $X = \mathsf{out}(x)$, $Y = \mathsf{in}(y)$ and $Z = X \cap Y$. If $X \neq Y$, $|Z \cap \mathcal{TEP}| > 1$ and $range(\mathsf{Att}) \cap Z = \emptyset$, a subset of $\mathcal{PS}$ can be identified as $\mathcal{Q} =(\mathcal{O}_q, \mathcal{F}_q, \emptyset, \emptyset)$ where $\mathcal{O}_q = \{x, y\} \cup X \cup Y$and $\mathcal{F}_q = \mathcal{F} \cap (\mathcal{O}_q \times \mathcal{O}_q)$. $\mathcal{Q}$ is called a quasi-FLOW-pattern and can be converted to $\mathcal{Q}' = (\mathcal{O}'_q, \mathcal{F}'_q, \emptyset, \emptyset)$ where
   - $\mathcal{O}'_q = \mathcal{O}_q \cup \{x', y'\}$,
   - $\mathcal{F}'_q = (\mathcal{F}_q \setminus ((\{x\} \times Z) \cup (Z \times \{y\}))) \cup \{(x, x'), (y', y)\} \cup (\{x'\} \times Z) \cup (Z \times \{y'\})$ and
   - $\mathcal{Q}'$ contains a FLOW-pattern $\mathcal{C} = (Z \cup \{x', y'\}, (\{x'\} \times Z) \cup (Z \times \{y'\}), \emptyset, \emptyset)$.

(b) Let $x \in \mathcal{G}^{ID}$, $y \in \mathcal{G}^{IM}$, $X = \mathsf{out}(x)$, $Y = \mathsf{in}(y)$ and $Z = X \cap Y$. If $X \neq Y$, $|Z \cap \mathcal{TEP}| > 1$ and $range(\mathsf{Att}) \cap Z = \emptyset$, a subset of $\mathcal{PS}$ can be identified as $\mathcal{Q} =(\mathcal{O}_q, \mathcal{F}_q, \mathsf{Cond}_q, \emptyset)$ where $\mathcal{O}_q = \{x, y\} \cup X \cup Y$, $\mathcal{F}_q = \mathcal{F} \cap (\mathcal{O}_q \times \mathcal{O}_q)$ and $\mathsf{Cond}_q = \mathsf{Cond}[\mathcal{F}_q]$. $\mathcal{Q}$ is called a quasi-SPECIAL-FLOW-pattern and can be converted to $\mathcal{Q}' = (\mathcal{O}'_q, \mathcal{F}'_q, \mathsf{Cond}'_q, \emptyset)$ where
   - $\mathcal{O}'_q = \mathcal{O}_q \cup \{x', y'\}$,
   - $\mathcal{F}'_q = (\mathcal{F}_q \setminus ((\{x\} \times Z) \cup (Z \times \{y\}))) \cup \{(x, x'), (y', y)\} \cup (\{x'\} \times Z) \cup (Z \times \{y'\})$,
   - $\mathsf{Cond}'_q(w, z) = \begin{cases} \mathsf{true} & \text{if } w = x \text{ and } z = x' \\ \mathsf{Cond}_q & \text{otherwise} \end{cases}$
   - $\mathcal{Q}'$ contains a FLOW-pattern $\mathcal{C} = (Z \cup \{x', y'\}, (\{x'\} \times Z) \cup (Z \times \{y'\}), \mathsf{Cond}_q, \emptyset)$.

(c) Let $x \in \mathcal{G}^D$, $y \in (\mathcal{G}^M \cup \mathcal{G}^{IM})$, $X = \mathsf{out}(x) \setminus \{y\}$, $Y = \mathsf{in}(y)$ and . If $Z = X \cap Y$, $X \subset Y$, $|Z \cap \mathcal{TEP}| + |Y \cap \{x\}| > 1$ and $range(\mathsf{Att}) \cap Z = \emptyset$, a subset of $\mathcal{PS}$ can be identified as $\mathcal{Q} = (\mathcal{O}_q, \mathcal{F}_q, \mathsf{Cond}_q, \emptyset)$ where $\mathcal{O}_q = \{x, y\} \cup X \cup Y$, $\mathcal{F}_q = \mathcal{F} \cap (\mathcal{O}_q \times \mathcal{O}_q)$ and $\mathsf{Cond}_q = \mathsf{Cond}[\mathcal{F}_q]$. $\mathcal{Q}$ is called a quasi-IF-pattern and can be converted to $\mathcal{Q}' = (\mathcal{O}'_q, \mathcal{F}'_q, \mathsf{Cond}_q, \emptyset)$ where

- $\mathcal{O}'_q = \mathcal{O}_q \cup \{y'\}$
- $\mathcal{F}'_q = (\mathcal{F}_q \setminus (Z \times \{y\})) \cup \{(y', y)\} \cup (Z \times \{y'\})$ and
- $\mathcal{Q}'$ contains an IF-pattern $\mathcal{C} = (Z \cup \{x, y'\}, (\{x\} \times Z) \cup (Z \times \{y'\}), \mathsf{Cond}_q, \emptyset)$.

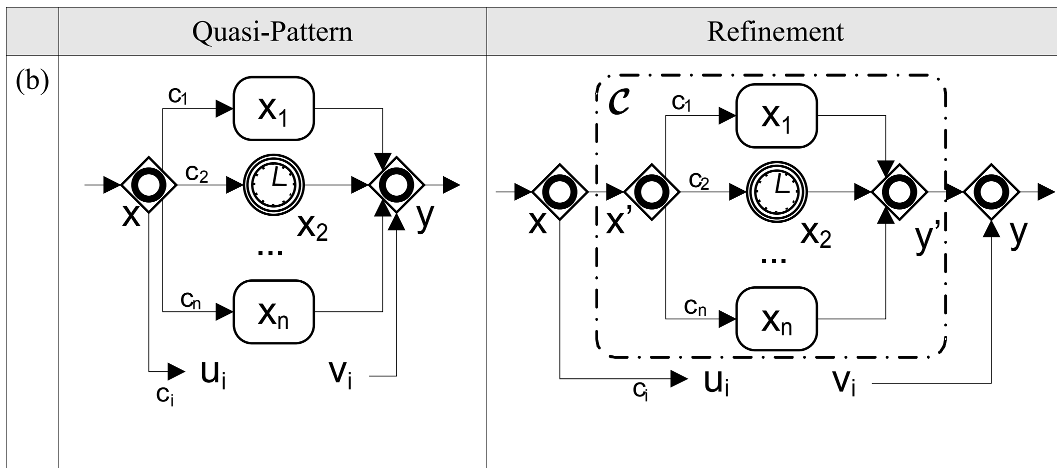
(d) Let $x \in \mathcal{G}^V$, $y \in (\mathcal{G}^M \cup \mathcal{G}^{IM})$, $X = \bigcup_{z \in \mathsf{out}(x)} \{\mathsf{succ(z)}\} \setminus \{y\}$, $Y = \mathsf{in}(y)$ and $Z = X \cap Y$. If $X \subset Y$, $|Z \cap \mathcal{TEP}| + |Y \cap \bigcup_{z \in \mathsf{out}(x)} \{x\}| > 1$ and $range(\mathsf{Att}) \cap Z = \emptyset$, a subset of $\mathcal{PS}$ can be identified as $\mathcal{Q} = (\mathcal{O}_q, \mathcal{F}_q, \emptyset, \emptyset)$ where $\mathcal{O}_q = \{x, y\} \cup X \cup Y$, $\mathcal{F}_q = \mathcal{F} \cap (\mathcal{O}_q \times \mathcal{O}_q)$. $\mathcal{Q}$ is called a quasi-PICK-pattern and can be converted to $\mathcal{Q}' = (\mathcal{O}'_q, \mathcal{F}'_q, \emptyset, \emptyset)$ where

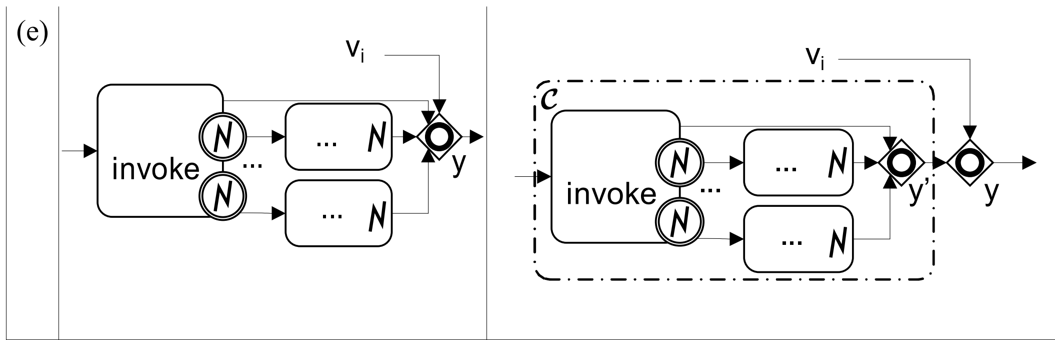
- $\mathcal{O}'_q = \mathcal{O}_q \cup \{y'\}$
- $\mathcal{F}'_q = (\mathcal{F}_q \setminus (Z \times \{y\})) \cup \{(y', y)\} \cup (Z \times \{y'\})$
- $\mathcal{Q}'$ contains a PICK-pattern $\mathcal{C} = (Z \cup \{x, y'\}, (\{x\} \times Z) \cup (Z \times \{y'\}), \emptyset, \emptyset)$

(e) Let $x \in (\mathcal{P}^S \cup \mathcal{T})$, $y \in (\mathcal{G}^M \cup \mathcal{G}^{IM})$, $Z = \{z \in \mathcal{E}^I_E | \mathsf{Att}(z) = x\}$, $X = \{x \in \mathcal{P}^F | \exists z \in Z : (z, x) \in \mathcal{F}\}$, $Y = \mathsf{in}(y)$. If $Z \neq \emptyset$, $y = \mathsf{out}(x)$ and $X \subset Y$, a subset of $\mathcal{PS}$ can be identified as $\mathcal{Q} = (\mathcal{O}_q, \mathcal{F}_q, \emptyset, \mathsf{Att}_q)$ where $\mathcal{O}_q = \{x, y\} \cup X \cup Y \cup Z$, $\mathcal{F}_q = \mathcal{F} \cap (\mathcal{O}_q \times \mathcal{O}_q)$ and $\mathsf{Att}_q = \mathsf{Att}[Z]$. $\mathcal{Q}$ is called a quasi-FAULT-HANDLER-pattern and can be converted to where $\mathcal{Q}' = (\mathcal{O}'_q, \mathcal{F}'_q, \emptyset, \mathsf{Att}_q)$

- $\mathcal{O}'_q = \mathcal{O}_q \cup \{y'\}$
- $\mathcal{F}'_q = (\mathcal{F}_q \setminus (X \times \{y\})) \cup \{(y', y)\} \cup (X \times \{y'\})$
- $\mathsf{Att}'_q = \mathsf{Att}_q$
- $\mathcal{Q}'$ contains a FAULT-HANDLER-pattern $\mathcal{C} = (X \cup Z \cup \{x, y'\}, (x \times y') \cup (X \times \{y'\}) \cup (\mathcal{F}'_q \cap (Z \times X)), \emptyset, \mathsf{Att}'_q)$.

The table below depicts examples for the quasi-special-flow and the quasi-fault-handler pattern as well as their corresponding refinement. Since different gateway types are allowed for the sink objects, the type of the additional gateways x' and y' is equivalent to the type of the gateways x and y respectively.

	Quasi-Pattern	Refinement
(b)		



The next pattern type, which the transformation is based on, is called generalized flow pattern. Definition 6 specifies this pattern in a formal way. It is taken from the original transformation and extended with the inclusive merge gateways.

Definition 6 (Generalized FLOW-pattern). Let $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c, \mathsf{Att}_c)$ be a component of a well-formed process $\mathcal{PS}$. $\mathcal{C}$ is a generalized FLOW-pattern iff:

– $\mathcal{C}$ contains no cycles,

– all the gateways in $\mathcal{C}$ are parallel fork, parallel join or inclusive merge gateways (i.e., $\mathcal{G}_c = \mathcal{G}_c^F \cup \mathcal{G}_c^J \cup \mathcal{G}_c^{IM}$) and

– there is no other component $\mathcal{C}' = (\mathcal{O}'_c, \mathcal{F}'_c, \mathsf{Cond}'_c, \mathsf{Att}'_c)$ such that $\mathcal{O}'_c \subset \mathcal{O}_c$

Ouyang et al. describes an algorithm in [ODvdAtH06] that maps this pattern onto a combination of BPEL4Chor structured activities (`flow` and `sequence`) with as few control links as possible. This mapping algorithm is used unchanged (except the permission of inclusive merge gateways).

Beside the pattern based transformation Ouyang et al. proposes a control link based translation in [ODvdAtH06]. The translation is applied if there is no component that matches any of the patterns above. It can be used for acyclic components that are sound and safe and do not contain event-based exclusive decision gateways. A component like this is called *synchronizing process component*. The formal definition for a synchronizing process component is not given in this thesis, because it can be found in [ODvdAtH06]. This definition does not need to be adapted for BPMN+.

The control link based translation transforms the sequence flows between all tasks and events into a set of control links within a `flow` element. Since it also allows exclusive decision gateways, it has to consider the conditions defined for the sequence flow connected to these gateways. In BPMN+, conditional branches of such a gateway should be evaluated in a specific order. In more detail, the first condition that evaluates to true determines the sequence flow that will be taken. The translation ensures that this semantic is preserved by refining the conditions of the flow. In BPMN+ and BPEL4Chor, the expression language for the transition conditions is not definite and conditions can even be omitted. The transformation cannot take into account all possible expression languages. That is why all conditions are treated as XPath boolean expressions (see [XPath99]), which is the default expression language in BPEL4Chor. Since the syntax of XPath boolean expressions is easy to understand, it can also be used for conditions formulated in plain English. Opaque expressions cannot be refined at all. So the conditions of the outgoing sequence flows of a data-based decision gateway can only be refined if each of them is not opaque. If there is an opaque condition, then the other conditions need to be opaque, too. In this case the refined condition is opaque as well.

For implementing the translation, Ouyang et al. defines the two functions PreTEC-Sets and PreTEC-SetsFlow. They are used to determine the transition conditions for the links

and the join conditions for the activities of the component. The PreTEC-SetsFlow function is adapted below to support inclusive gateways. Beside this modification, the translation can be performed as defined in [ODvdAtH06].

The function PreTEC-SetsFlow takes the source activity y and the target activity x of a flow as input. It generates a set of sets each containing the preceding tasks, events and/or conditions of the flow. The function RefinedCond returns the refined transition condition as described above. PreTEC-Sets produces the same output as PreTEC-SetsFlow but takes as input an activity.

```
1: function PreTEC-SetsFlow(y: Object, x:Object)
2: begin
3:   if y is a task, an event or a sub-process then
4:     PreTEC-SetsFlow:={{y}}
5:   else
6:     if y is a parallel fork gateway then
7:       PreTEC-SetsFlow:=PreTEC-Sets(y)
8:     else
9:       if y is a data-based exclusive decision gateway then
10:        PreTEC-SetsFlow:=AddCond(RefinedCond(y, x),PreTEC-Sets(y))
11:      else
12:        if y is an exclusive merge gateway then
13:          PreTEC-SetsFlow:=⋃_{z∈in(y)} PreTEC-SetsFlow(z, y)
14:        else
15:          if y is a parallel join gateway then
16:            PreTEC-SetsFlow:=∏_{z∈in(y)} PreTEC-SetsFlow(z, y)
17:          else
18:            if y is an inclusive decision gateway
19:              PreTEC-SetsFlow:=AddCond(Cond(y, x),PreTEC-Sets(y))
20:            else // i.e. y is an inclusive merge gateway
21:              PreTEC-SetsFlow:=⋃_{z∈in(y)} PreTEC-SetsFlow(z, y)
22: end
23:// Note that the above function makes use of the following auxiliary function:
24:// AddCond(b, {p_1, p_2, ..., p_n}) = {p_1 ∪ {b}, p_2 ∪ {b}, ..., p_n ∪ {b}}
```

As can be seen from the code above, the inclusive gateway is treated like an exclusive gateway. However the transition conditions do not need to be refined, because inclusive gateways do not define an evaluation order.

4.2.3 Mapping of Tasks, Events and Scopes

The tasks, events and scopes contained within the sequence flow of components are mapped to the BPEL4Chor basic activities during the folding of the components. The mapping of almost all tasks is straightforward. The table below shows the mapping of the tasks to their BPEL4Chor representation.

Only send tasks can be mapped to different BPEL4Chor representations. If a send task is connected with a message flow that leads to a service task and the outgoing message flows of the service task lead to activities that are located before the send task, it maps to a BPEL4Chor `reply`. Otherwise, the send task maps to an asynchronous BPEL4Chor `invoke`.

BPMN$^+$	BPEL4Chor
service task	synchronous *invoke*
receive task	receive
send task	asynchronous invoke or reply
assign task	assign
validate task	validate
empty task	empty
non-typed task	opaqueActivity

The mapping of events is straightforward as well. The table below shows the mapping of BPMN$^+$ events to BPEL4Chor activities. A message start event located in a message handler provides the information for the onEvent branch of the handler. A message intermediate event that is the successor of an event-based decision gateway is mapped to the onMessage branch of a pick if the component matches to a pick pattern. If a compensation intermediate event specifies the activity attribute that defines the scope to be compensated, it maps to compensateScope. An error intermediate event located in a fault handler is mapped to a rethrow if the event does not specify the error code attribute.

BPMN$^+$	BPEL4Chor
message start event	receive/ onEvent
timer start event	onAlarm
message intermediate event	*receive/ onMessage*
timer intermediate event	wait
compensation intermediate event	compensate/ compensateScope
error intermediate event	throw/ rethrow

Sub-processes of the type scope are mapped to BPEL4Chor scopes. The mapping of handlers is explained in 4.2.4.

Looping tasks and sub-processes are mapped to looping BPEL4Chor activities depending on the loop type. A multi-instance loop is mapped to a BPEL4Chor forEach. A standard loop is mapped to a while if the test time attribute has the value “Before” and it is mapped to a repeatUntil of the test time attribute has the value “After”. The actual task or scope that defines the loop is mapped as described above and added into the created BPEL4Chor looping element.

4.2.4 Mapping of Handlers

Handlers, except fault handlers, are not connected with the normal sequence flow of a process. Timer and message event handlers have no incoming or outgoing sequence flows, termination handlers have only one incoming sequence flow from the attached ter-

mination event and compensation handlers are connected with associations instead of sequence flows. That is why they are not mapped using patterns.

Timer and message event handlers are mapped directly before the sequence flow transformation starts. The triggering information is taken from the timer or message start event that is located within the event handler. The sequence flow within the event handlers is mapped using the extended sequence flow transformation again. Event handlers can also have attached intermediate events that lead to other handlers. If these are fault handlers they do not have to be merged with the normal sequence flow, since event handlers do not have outgoing sequence flow. That is why the handlers connected to event handlers can be mapped directly without using patterns. They will be generated as handlers within the `scope` of the BPEL4Chor event handler.

Termination and compensation handlers are mapped during the mapping of the task and sub-process they belong to. The actual sequence flow within these handlers is mapped using the extended sequence flow transformation again.

Since attached events are not allowed for pools or pool sets, the fault handlers of a process must be modeled using an additional scope. In this case the scope, the attached error events, the fault-handlers and the merging gateway are the only activities between the start and the end event of the process. Such a scope is not transformed to a BPEL4Chor `scope`. Instead of this, the content of the scope and the fault handlers are generated into the BPEL4Chor `process` element.

4.2.5 Variables

The generation of variables is based on the standard variable data objects located in the BPMN^{+} processes or sub-processes. The variable definitions are generated before the actual sequence flow is transformed. If standard variable data objects are located within a handler an additional `scope` element has to be created the variable definitions will be generated into.

The access to the variables from the process activities is generated during the mapping of the tasks, events and handlers. They can be determined from the associations between the variable data objects to the activities. The direction of the association is used to determine if the data object represents an input or output variable. Since variable data objects can be associated with activities from different components, the variable data objects are not removed during the folding of a component.

4.2.6 Transformable Process Models

A process must fulfill at least Definition 1 and Definition 2 given above to be transformable to BPEL4Chor. However, these definitions are necessary, but not sufficient for a transformable process. They still allow unstructured cycles and processes that are not safe and sound. Processes with these characteristics cannot be transformed to BPEL4Chor with this transformation. This section will discuss what needs to be checked in addition to the definitions to assure that a process can be transformed to BPEL4Chor.

In [DDO07] a mapping of BPMN to Petri nets is defined. In this way static analysis like the soundness and safeness of BPMN^{+} models can be done using existing techniques for Petri nets (see [vdAa97]). This mapping does not deal with (i) multi-instance activities, where the number of instances is only known at runtime, (ii) exception handling in the context of sub-processes that are executed multiple times and (iii) inclusive gateways. These limitations will be discussed in the following:

As stated in [DDO07] multi-instance activities can be mapped to a pair of parallel split and join gateways enclosing a certain number of activities that represent the activity instances. This is similar to the flow pattern that is known to be sound and safe independent from the number of enclosed activities. Thus, the number of instances for a looping activity does not affect the soundness and safeness of a process. So we can omit this problem and assume a pre-defined number of instances for the mapping. Exception handling is allowed only if it is modeled as specified in the fault-handler-pattern. This allowed modeling implies that the usage of fault handlers is always sound and safe. Thus, the exception handling can be omitted in the mapping, because the right usage can be checked before the mapping to Petri nets. However, the soundness and safeness of the flow contained in a sub-process (e.g. the fault handler) needs to be checked mapping it to an independent Petri net.

Having introduced inclusive gateways in the transformation to BPEL4Chor, they must be introduced in the mapping to Petri nets as well. The mapping of an inclusive decision gateway is straightforward and illustrated in Figure 4.3. The non-local semantic of an inclusive merge gateway cannot be mapped to a standard Petri net that has a local nature of transitions. In [LSW97] a class of colored Petri nets called *Boolean nets* was introduced. These nets are powerful enough to represent the semantic of inclusive merge gateways using a mechanism similar to the dead-path elimination in workflow systems (see [LA94]). A drawback of Boolean nets is that loops must be restricted. A loop must have an exclusive merge gateway as entry point and an exclusive decision gateway as exit point. These will be mapped to one place in the resulting Boolean net. Inclusive gateways are not allowed as entry point like in the looping patterns described in Definition 4. However, in these patterns the inclusive merge acts as exclusive merge. That is why it can be mapped like an exclusive merge gateway in this special case.

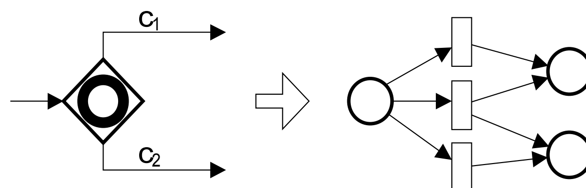


Figure 4.3: Mapping inclusive decision gateway onto Petri net modules [WEvdA+05]

The transformation of BPMN$^+$ to BPEL4Chor does only allow structured cycles as specified in the while-, repeat- and repeat-while-patterns. Moreover, the mapping of the model to a Boolean net assumes that contained cycles are structured. So the process is not transformable if it contains unstructured cycles. To determine whether a cycle is valid, the entry and exit points must be determined. If there is more than one entry or exit point, the cycle is unstructured and cannot be transformed. The exit point of a cycle has to have exactly two outgoing sequence flows: one that leads to back to the entry point and one to leave the cycle. This restriction is specified in the well-structured cycle patterns – the only way to transform cycles to BPEL4Chor with the described transformation.

In section 4.2.1 the restrictions for multiple start and end events were already discussed (e.g. only multiple message start events allowed). These restrictions must be checked to ensure the start and end events can be combined.

The event-based exclusive decision gateway can only be transformed using the (quasi) pick-pattern. Thus, it can only be transformed if all outgoing sequence flows lead to the same exclusive or inclusive merging gateway.

4.2.7 Transformation Implementation Considerations

The content of the BPEL4Chor processes is generated from the pools and pool sets of the BPMN+ diagram using the extended transformation explained above. The content of sub-processes located in the pools and pool sets is also generated with this transformation. These processes or sub-processes must be well-formed processes as specified in Definition 2. For each well-formed process the transformation executes the following steps:

1. Create variable definitions from the variable data objects (see 4.2.5)

2. Create BPEL4Chor event handlers from the BPMN+ event handlers (see 4.2.4)

3. Remove multiple start and end events (see 4.2.1)

4. While the process is not trivial (see Definition 7)

4.1. Determine the next component in the process. The mapping starts with the fault-handler-patterns. If there are no fault-handler patterns, left the sequence-flow-patterns are processed and then the other well-structured components. If no well-structured component is left, continue with the quasi-structured components, then the generalized-flow-pattern. If none of the components above was found, check for a synchronizing process component.

4.1.1. If the component is well-structured component then fold it (see Definition 8).

4.1.2. If the component is quasi-structured then refine it according the definition of the pattern.

4.1.3. If the component matches a generalized-flow-pattern, apply the algorithm defined in [ODtHvdA07].

4.1.4. If the component is a synchronizing process, apply the algorithm defined in [ODvdAtH06].

4.1.5. If no component found return an error.

5. Map the trivial process

The characteristics of a trivial process are given in Definition 7. If the process has a message triggered start event, the start event is mapped in addition to the activity x. There is generated a `sequence` element, containing the mapping of the start event and the mapping of the activity. If there is no other activity, the generated control flow contains only the mapping of the start event without the additional `sequence`. Since the transformation is also used for the sequence flow within sub-processes the start events of those have to be considered, too. The message start event of a message event handler is not interpreted as a `receive`. That is why it will not be treated as in a process. Other sub-processes are not allowed to contain message triggered start events at all.

Definition 7 (Trivial well-formed process): Let $\mathcal{PS} = (\mathcal{O}, \mathcal{F}, \mathsf{Cond}, \mathsf{Att})$ be a well-formed process. $\mathcal{PS}$ is trivial iff:

– $\{s\} = \mathcal{E}^S \wedge \{e\} = \mathcal{E}^E$

– $dom(\mathsf{Att}) = \emptyset$

– if $(s, e) \in \mathcal{F} \Rightarrow s \in \mathcal{E}^S_M$

– if $\exists x \in \mathcal{O} \setminus \mathcal{E}^E$ with $(s, x) \in \mathcal{F} \Rightarrow (x, e) \in \mathcal{F}$

The folding of a well-structured component is defined in the original transformation of Ouyang et al. This definition has to be extended, because attached fault events are introduced in the definition of a well-formed process.

Definition 8 (Fold): Let $\mathcal{PS} = (\mathcal{O}, \mathcal{F}, \mathsf{Cond}, \mathsf{Att})$ be a well-formed process and $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c, \mathsf{Att}_c)$ be a component of $\mathcal{PS}$. The function Fold replaces $\mathcal{C}$ in $\mathcal{PS}$ by a blank task object $t_c \notin \mathcal{O}$, i.e., $\mathsf{Fold}(\mathcal{PS}, \mathcal{C}, t_c) = (\mathcal{O}', \mathcal{F}', \mathsf{Cond}', \mathsf{Att}')$ with:

- $\mathcal{O}' = (\mathcal{O} \setminus \mathcal{O}_c) \cup \{t_c\}$,
- $\mathcal{T}_c$ is the set of tasks in $\mathcal{C}$, i.e., $\mathcal{T}_c = \mathcal{O}_c \cap \mathcal{T}$,
- $\mathcal{T}' = (\mathcal{T} \setminus \mathcal{T}_c) \cup \{t_c\}$ is the set of tasks in $\mathsf{Fold}(\mathcal{PS}, \mathcal{C}, t_c)$,
- $\mathcal{T}^{R'} = (\mathcal{T}^R \setminus \mathcal{T}_c)$ is the set of receive tasks in $\mathsf{Fold}(\mathcal{PS}, \mathcal{C}, t_c)$, i.e., t_c is not a receive task,
- $\mathcal{F}' = (\mathcal{F} \cap (\mathcal{O} \setminus \mathcal{O}_c \times \mathcal{O} \setminus \mathcal{O}_c)) \cup \{(\mathsf{entry}(\mathcal{C}), t_c), (t_c, \mathsf{exit}(\mathcal{C}))\}$
- $\mathsf{Cond}' = \begin{cases} \mathsf{Cond}\,[\mathcal{F}'] & \text{if } \mathsf{entry}(\mathcal{C}) \notin (\mathcal{G}^D \cup \mathcal{G}^{ID}) \\ \mathsf{Cond}\,[\mathcal{F}'] \cup \{((\mathsf{entry}(\mathcal{C}), t_c), \mathsf{Cond}(\mathsf{entry}(\mathcal{C}), i_c))\}, & \text{otherwise} \end{cases}$
- $\mathsf{Att}' = \mathsf{Att}[\mathcal{O}']$

4.3 Generating the BPEL4Chor Topology

It is not a long time ago that BPEL4Chor was developed. Thus, there is no transformation described yet that maps a BPMN diagram to a BPEL4Chor participant topology. That is why a transformation will be introduced in this section that generates the BPEL4Chor topology from a BPMN$^+$ diagram.

Since the transformation of the BPEL4Chor processes removes activities from the process (e.g. folding of patterns), the transformation of the topology has to be executed before the transformation of the processes. The activities within the diagram are needed for the generation of the message flow and for determining the type of the participants.

The transformation of the topology is done in three steps:

1. Generate the participant types
2. Generate the participants
3. Generate the message links

These steps are described in detail in the next sections.

4.3.1 Participant Types

The participant types of the BPEL4Chor topology can be generated directly from the pools and pool sets contained in the BPMN$^+$ diagram. Each pool or pool set represents a specific participant type. The name of the participant type corresponds to the name of the pool or pool set. The referenced participant behavior description corresponds to the name of the process that will later be generated with the transformation from 4.2. Listing 4.1 shows the generated participant types of the example illustrated in Figure 2.15.

```
<participantTypes>
    <participantType name="Client"
        participantBehaviorDescription="c:client"/>
    <participantType name="Bookshop"
        participantBehaviorDescription="bs:bookshop" />
    <participantType name="Supplier"
        participantBehaviorDescription="sup:supplier" />
</participantTypes>
```

Listing 4.1: Example for generated participant types

4.3.2 Participants

The BPEL4Chor participant hierarchy with the participant sets and participant references is generated from the pools, participant reference data objects and participant set data objects located in the diagram.

Participant references and participant sets are identified by their name, e.g., participant reference data objects with the same name represent the same participant reference. A pool represents a participant reference and defines the name of this reference in an additional attribute.

The participants are generated in a specified order.

1. Generate participant references from pools
2. Generate participant sets from the participant set data objects
3. Generate the remaining participant references from the participant reference data objects

In contrast to pool sets, pools represent participants that do not occur in a participant set. That is why first of all the participant reference for each pool will be generated. The type of the participant reference corresponds to the name of the pool.

After that the participant sets will be created from the participant set data objects. The contained participant sets and participant references are created, too. The creation starts with the top-level participant set, i.e., a participant set that is not contained in any other set. All data objects that represent a top-level participant set are not connected with an association that leads to another participant set.

The top-level participant must reference a participant type in the BPEL4Chor topology. There are several possibilities to determine this type. However, these possibilities always lead to the identification of the participant type of a participant reference. The participant type of a participant reference can be determined as follows.

Determine participant type of a participant reference:

From associations: Participant reference data objects can be associated with communicating activities to illustrate the sender and the receiver of a message. A directed association from a participant reference data object to a sending task (service task, send task) expresses that the message is sent to this participant. The pool or pool set that contains the receiving activity of the message represents the type of this participant reference. Figure 4.4 shows an example for this situation. The type of the participant reference “receiver” is represented by the pool set “Y” that contains the receiving intermediate event. A directed association from a receive task or message intermediate event to the participant reference data object expresses that the sender of the message will be bound to this partici-

pant reference. So the pool or pool set that contains the sending activity of the message represents the type of the participant. This applies also for a directed association from the participant reference data object to a receiving activity. In Figure 4.4 the type of the participant "sender" can be determined in this way. The type is represented by the pool set "X" containing the sending activity.

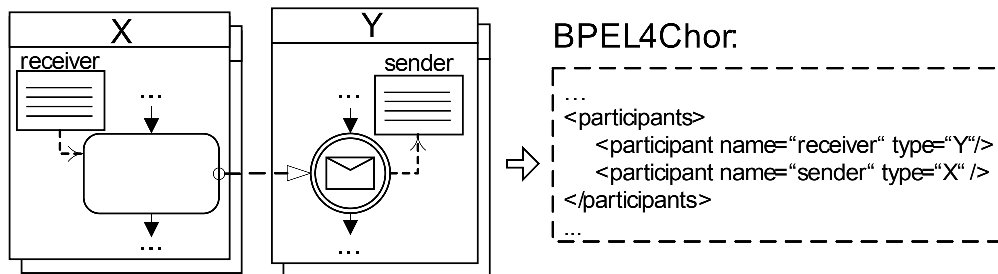


Figure 4.4: Generating the type of participant references from associations

From CopyTo attribute: Participant reference data objects can specify a CopyTo attribute if they are passed over a message flow. This attribute denotes the participant reference the passed participant reference is copied to when the message arrives. The types of these two participant references must be equal. That is why the type of a participant reference can be determined from the participant reference it is copied to.

From other participant references: There may be multiple participant reference data objects representing the same participant reference. If the type of a participant reference could not be determined with one of the possibilities above, the type may be generated from one of the other participant reference data objects that have the same name.

Now that the participant type for a participant reference can be identified, this can be used for determining the type of a participant set.

Determine participant type of a participant set:

From associations: If the participant set data object has a directed association to a multi-instance loop, the loop is iterating over the participants contained in the set. In this case the participant type can be determined from the participant reference that acts as loop counter. This participant reference data object is associated with the multi-instance loop, too. The participant reference must be contained in the set and therefore has the same participant type. If there is a directed association from a receive task or message intermediate event to the participant set data object, the sender of the message will be added to the participant set. In this case there must be participant reference data object associated with the receiving activity, too. This participant reference data object expresses the actual reference the sender of the message will be bound to. The participant reference must be contained in the set and therefore has the same participant type. In Figure 4.5 this can be seen for the participant set "senders". The receiving activity, "senders" is associated with a looping task. This task has another association to the participant reference "bindTo". "Senders" contains the possible senders of the message. The actual sender will be bound to "bindTo". So "bindTo" is contained in the "senders" participant set. The type of "senders" corresponds to the type of "bindTo" and the type of "bindTo" can be determined as described above.

From CopyTo attribute: Like participant reference data objects, participant set data objects can specify a CopyTo attribute if they are passed over a message flow. This attribute denotes the participant set the passed participant set is copied to when the message ar-

rives. The types of these two participant sets must be equal. That is why the type of a participant set can be determined from the participant set it is copied to.

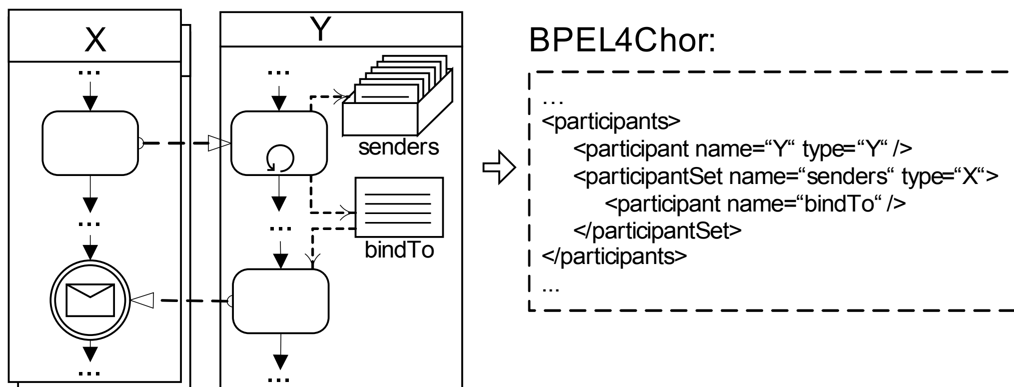


Figure 4.5: Generating the type of participant sets from associations

From contained references and sets: The containment of a participant references and sets in another participant set can be modeled using directed associations in BPMN+. The type of the contained participant references and sets must be equal to the type of the participant set they are contained in. That is why the type of a participant set can be determined from the types of its contained participant references and sets.

From other participant sets: There may be multiple participant set data objects representing the same participant set. If the type of a participant set could not be determined with one of the possibilities above, the type may be generated from one of the other participant set data objects that have the same name.

Containment

After a top-level participant set is generated, the contained participant sets and references can be created. Contained participant sets can only be determined from the directed associations between two participant set data objects. The containment of participant references in sets can be modeled and determined in this way, too. The type of contained participant references and sets is not generated, since they are of the same type as the parent participant set.

There are situations which implicitly express a containment of a participant reference in a participant set. If the participant set data object is associated with multi-instance loops, the loop counters are contained in the set. Figure 4.6 illustrates this situation. If the looping activity is a task, the loop counter participant reference does not need to be modeled explicitly. Since there are no other activities in the loop, the counter can only be associated with this task. So the data object can be inserted automatically during the transformation.

If the participant set data object is the target of an association from a receiving activity, there must be a participant reference data object associated with the receiving activity, too. This reference is contained in the set. This type of implicit containment can be seen in Figure 4.5. If this participant reference is not used by the following activities, it does not need to be modeled explicitly. In this case this participant reference data object will be inserted automatically during the transformation.

At the end of the participant transformation the remaining participant references will be generated. They are not contained in a participant set, so the type must be determined as described above.

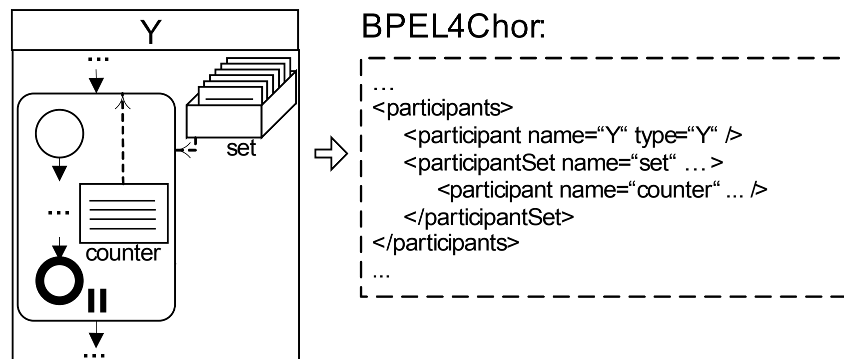


Figure 4.6: Implicit containment

Attributes

The BEL4Chor `forEach` attribute of participant references and participant sets is determined from the multi-instance loops the data objects are associated with. A participant reference can only be associated with one multi-instance loop. The attribute value is generated from the name of this loop. Participant set data objects can be associated with more than one multi-instance loop. In this case the name of each loop is added to the attribute value.

Since there may be multiple data objects representing the same participant, the values of the selects attribute may be distributed to more than one participant reference data object. That is why the attribute value is retrieved from all data objects with the same name.

4.3.3 Message Links

The message links of the BPEL4Chor topology are generated from the message flows in the BPMN$^+$ diagram. BPEL4Chor allows multiple sending activities for one receiving activity. That is why message flows with the same target are generated to one message link with multiple sending activities. If there are several receiving activities for the same sending activity (e.g. through branching on the receiver's side), then all these receiving activities must have the same name. Thus, message flows with the same source are only generated to one message link if the name of the target activities is equal. Otherwise, the topology cannot be transformed.

The `sendActivity` of a message link is generated from the source activities of the appropriate message flows and the name of the `receiveActivity` is determined from the target activities.

If the sending activity is located in a pool, the pool represents the sending participant. Otherwise, the sender is determined from the data object associated to the target activity of the message flow. If there is only a single participant reference data object associated with the target activity, a single `sender` will be generated for the message link. If there is a directed association from the target activity to a participant set data object, the message can be sent from an arbitrary sender. In this case the value of the `senders` attribute is equal to the name of the participant set data object. The actual sender will be added to this set. Multiple `senders` can also be generated from multiple participant reference data objects with an association to the target activity. The names of these data objects build the he `senders` attribute value. Moreover, the `bindSenderTo` attribute of the message link must be set in the case of multiple senders. The attribute value is determined from the participant reference data object with a directed association from the target activity.

A message link may be generated from message flows with different target activities. These target activities represent the same receiving activity of the message link. That is why the associated data objects must be equal (equal names). An example for this is shown in Figure 4.7. Both participant reference data objects associated to the receiving activities "receive" have the name "sender".

A message link may be generated from message flows with different source activities. There can only be one receiver defined for a message link. That is why the data objects associated to the source activities must have equal names. This is illustrated in Figure 4.8. The sending activities "invoke1" and "invoke2" are associated with the same participant reference data object.

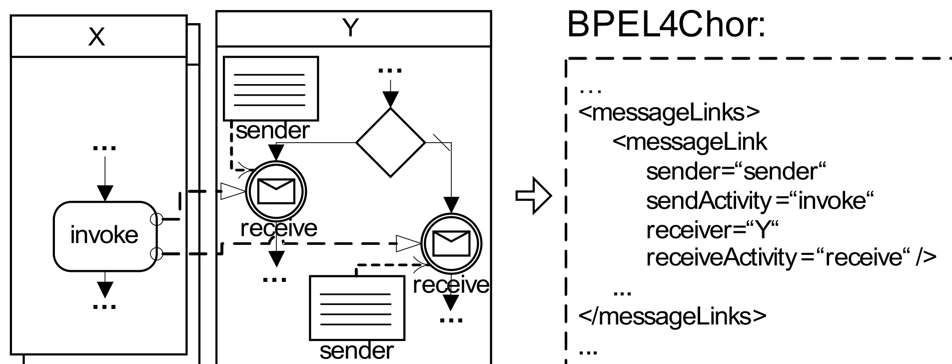


Figure 4.7: Message flows with the same source generated to one message link

The `receiver` of a message link can be determined from the pool the receiving activity is located in. If the receiving activity is located in a pool set, the `receiver` must be generated from the data object associated to the source activity.

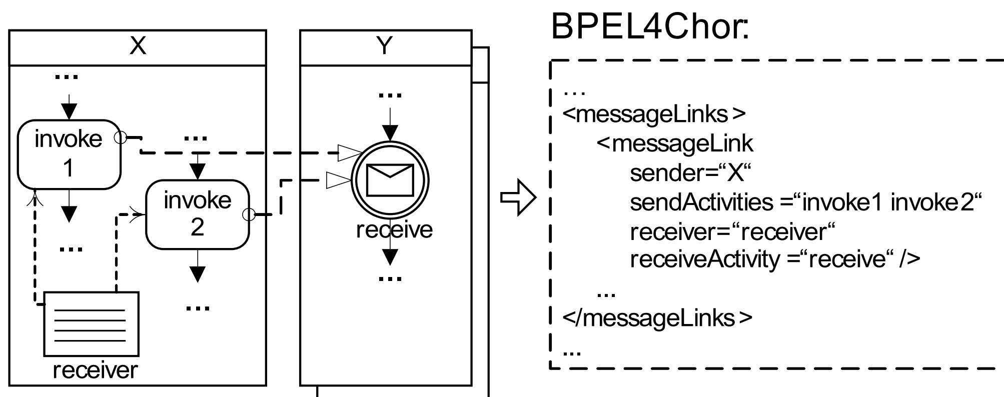


Figure 4.8: Message flows with the same target generated to one message link

The participant references and sets passed over a message link are expressed in BPMN+ using data objects associated to the message flow with a non-directed association. For each of these references and sets, the name of the data object is added to the attribute value of the `participantRefs` attribute. The CopyTo attribute, which can be defined for the data objects, is used to determine the value of the message link's `copyParticipantRefsTo` attribute. The order of the values in this attribute must correspond to the order of the participants in the `participantRefs` attribute. That is why all or none of the data objects must specify the CopyTo attribute. If the message link is genera-

ted from multiple message flows, these message flows must be associated with data objects that have the same names.

The name of the message link is equal to the name of the message flow and the name of the passed message is equal to the message name defined for the message flow. These attributes must be equal for message flows with the same source or the same target.

4.4 Implementation of the Transformation

The transformation depicted above should be independent from the editor that is used to model the BPMN$^+$ diagram. That is why the abstract transformation described in the chapter above is implemented as Java web service. In this way it can be called from any editor with access to the web service.

The input format of the transformation is XPDL4Chor that was introduced in chapter 2.4. So an editor calling the transformation has to convert his internal BPMN$^+$ data model to XPDL4Chor to call the transformation.

4.4.1 The Web Service

The port type of the web service does only provide one operation – the transformation of BPMN$^+$ diagrams to BPEL4Chor. The operation takes two parameters as input. The first one is the BPMN$^+$ diagram in XPDL4Chor serialized as string. In addition to this, the second parameter specifies whether the XPDL4Chor input should be validated against the XPDL4Chor schema before the transformation starts. The validation should be disabled if the XPDL4Chor schema file cannot be reached from the web service's environment.

The output of the transformation is an array of strings. This array contains the generated BPEL4Chor topology and the generated BPEL4Chor participant behavior descriptions. Each of them is serialized as string. The first element of the response array is the serialized topology and the other elements are the serialized participant behavior descriptions. If an error occurred during the transformation of a process or the topology, the response array contains the error description instead of the serialized BPEL4Chor document at the corresponding array index.

4.4.2 Architecture Overview

Figure 4.9 depicts the three main components of the implemented transformation: The *model* package, the *parser* package and the *transformation* package. In addition to these packages, the *util* package contains some helper classes for handling lists, XML files and BPEL files.

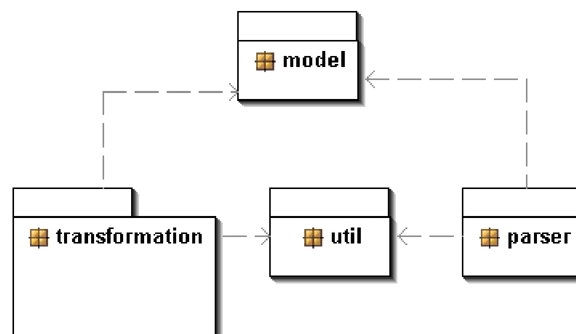


Figure 4.9: Main packages of the transformation web service

The *model* package contains the classes that provide all the information about the BPMN+ diagram. The objects and their attributes reflect the XPDL4Chor representation of the BPMN+ diagram. The *parser* package contains the classes that translate the information from the XPDL4Chor input into the internal model. The model itself is independent from the parser. In this way the current parser can be exchanged by other parsers to support different input formats. The *transformation* package holds the classes that do the actual transformation.

5 Editor Extension

The last challenge of this diploma thesis is to implement the extension of an existing BPMN editor with BPMN+ elements and the connection of this editor to the transformation.

In [BFV07] existing BPMN editors were already evaluated regarding their modeling abilities and extensibility. Most of these editors, e.g. Borland Together[1] or iGrafx FlowCharter[2], provide a mechanism to introduce new elements. Some editors even allow adding new functionality using a special API (e.g. Visual Paradigm Business Process Visual Architect[3]). Unfortunately nearly all of these editors are not free such they cannot be used for this thesis. Only the Soyatec eBPMN Designer[4] and the Oryx Business Process Editor[5] are available for this diploma thesis and provide the needed extensibility mechanism. Since the Oryx editor has a generic implementation that allows an easy extension of new graphical elements and functionality, it is more applicable for the extension with BPMN+ elements.

In the next sections the extension of the Oryx editor will be specified. First of all the implementation of the editor is depicted in more detail. After that the extension of the editor with the BPMN+ elements is described. And at the end of this chapter it will be explained how the BPEL4Chor transformation is called from the editor.

5.1 Oryx Editor

Oryx was developed as a bachelor's project at the Hasso Plattner Institute at the University of Potsdam. It is a web-based business process editor that already comes with a support for modeling BPMN diagrams.

It is implemented using JavaScript. It uses the JavaScript framework Prototype[6] that adds additional JavaScript functionality like class-driven development and an Ajax library. The Ajax library is used to send HTTP requests to the web server and to evaluate the answer without reloading the web page. To ease the creation of the user interface, the Ext[7] library is used. This library provides JavaScript classes for the creation of web applications with a graphical user interface that is similar to desktop applications.

To enable the editor to understand different notations Oryx introduces the definition of stencil sets. Every stencil set represents a graphical notation that can be used in the editor. A stencil set consists of stencils and rules that are specified in the JSON[8] file (JavaScript Object Notation). A stencil in the set represents a graphical diagram element. The graphical representation of a stencil is defined using Scalable Vector Graphics (SVG). SVG is a W3C standard [SVG03] to describe vector graphics in XML. In these files graphical objects like rectangles, cycles or polygons are represented as XML tags with certain attributes. In addition to the graphical issue, a stencil can have special properties that are defined in the JSON file, too. These properties can be of the type: "Boolean", "Choi-

1 See http://www.borland.com/us/products/together/
2 See http://www.igrafx.com/
3 See http://www.visual-paradigm.com/product/bpva/
4 See http://www.soyatec.com/ebpmn/
5 See http://www.oryx-editor.org
6 See http://www.prototypejs.org/
7 See http://extjs.com/
8 See http://json.org/

ce", "Color", "DateTime", "Float", "Integer", "String" and "Url". During the development of the process diagram the modeler can assign values to these properties for every graphical element.

The model data, which is generated this way, is embedded into the page using eRDF[1]. This is the underlying data layer the model data gets serialized to. More information about the storage of the model data can be found in [Czuc07].

For adding new functionality, the editor provides a plugin mechanism. Functionality like the buttons in the toolbar (at the top in Figure 5.1) or the property window (at the right in Figure 5.1) is introduced by a plugin. Even the toolbar itself is a plugin. There is a uniform interface for plugins that eases the creation of new plugins to add new functionality to the editor.

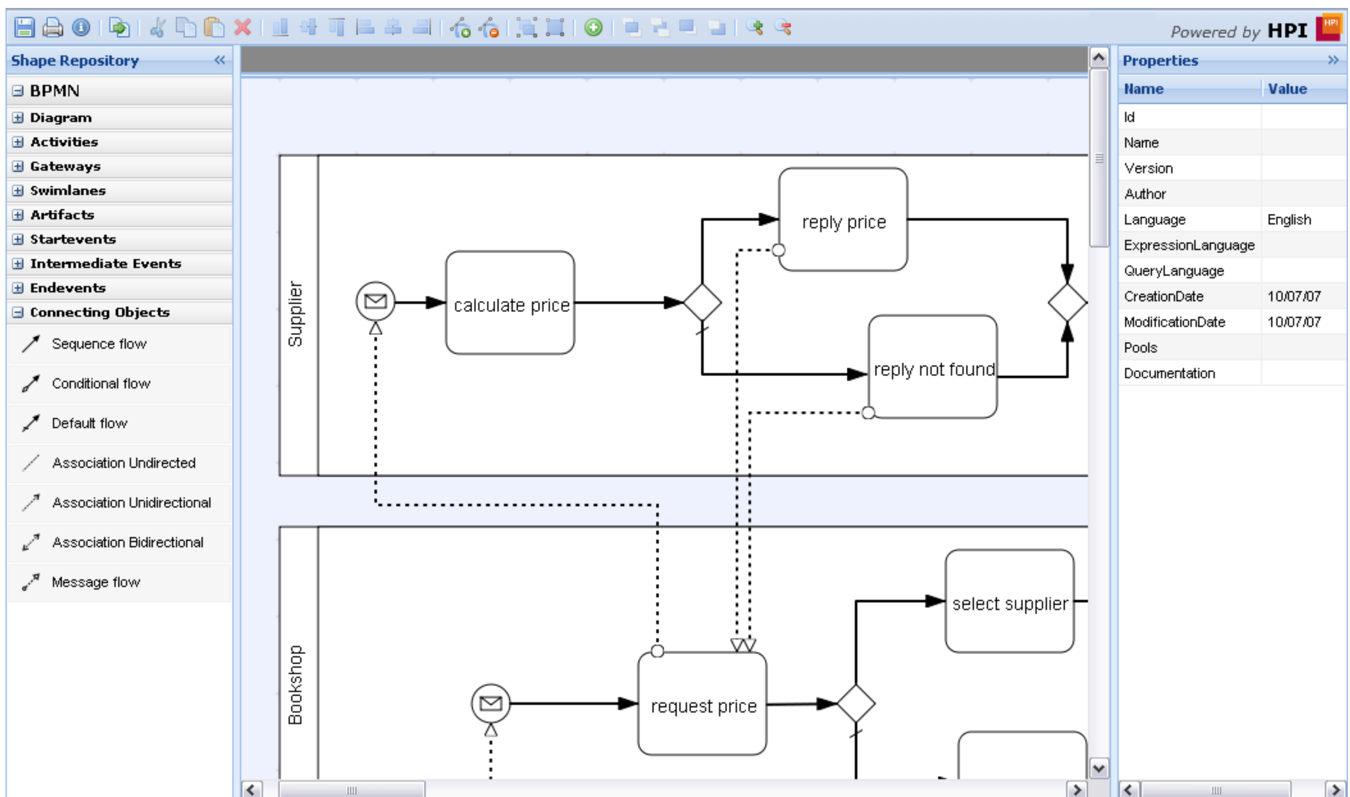


Figure 5.1: User interface of the Oryx editor

More information about the implementation of the editor and the usage of the technologies mentioned above can be found in [Tsch07].

5.2 BPMN+ Stencil Set

As mentioned above, Oryx introduces stencil sets to define graphical elements and their properties. Since Oryx already provides a BPMN stencil set, it is not necessary to start from scratch. Instead, the existing BPMN stencil set is be adapted to a BPMN+ stencil set in the following.

For this purpose, the SVG representation of new elements like pool sets, participant reference or participant set data objects is created. An example for the representation of a

1 See http://research.talis.com/2005/erdf/wiki/Main/RdfInHtml

pool is shown in Listing 5.1. It can be seen that Oryx has extended the SVG namespace with additional elements and attributes. For example, the `magnets` element was introduced to define points on a node where other nodes or edges can connect to. The `resize` attribute tells the editor that this element can be resized. The `rect` elements define the border and content of the pool set shape and the `text` element defines where the name of the pool should be located.

```
<svg   xmlns="http://www.w3.org/2000/svg"
       xmlns:oryx="http://www.b3mn.org/oryx"
       width="515" height="315" version="1.0">
    <oryx:magnets>
        <oryx:magnet oryx:cx="0" oryx:cy="156"
            oryx:anchors="left" />
        ...
    </oryx:magnets>
    <g pointer-events="fill" >
        <rect  id="border_shade" x="15" y="0" width="500"
               height="300" oryx:resize="vertical horizontal"
               stroke="black" stroke-width="1" fill="none" />
        <rect  id="c_shade" x="15" y="0" width="500"
               height="300" oryx:resize="vertical horizontal"
               stroke="none" fill="white" />
        <rect  id="c" x="0" y="15" width="500" height="300"
               oryx:resize="vertical horizontal" stroke="none"
               fill="white" />
        <rect  id="border" x="0" y="15" width="500"
               height="300" oryx:resize="vertical horizontal"
               stroke="black" stroke-width="1" fill="none" />
        <text  x="10" y="140" font-size="14" id="text_v"
               oryx:align="middle center" oryx:rotate="270"
               fill="black" stroke="black"/>
    </g>
</svg>
```

Listing 5.1: SVG file for a BPMN$^+$ pool set

The attributes of the BPMN$^+$ elements are defined as stencil properties in the JSON file of the stencil set. These properties are defined according to the list of attributes from chapter 2.3. Properties can only be specified for graphical elements with a shape that can be displayed in the diagram. Although a process is a BPMN$^+$ element, it does not have a certain shape. That is why the process properties must be defined for the pool or pool set the process belongs to.

The attributes of the elements are static and cannot be changed dynamically in the property window. That is why elements that are different in some attributes (e.g. the different task types) must be listed separately in the stencil set. In addition to this, the different gateway types are subdivided into split and merge gateways. In this way the different restrictions regarding the incoming and outgoing sequence flows can be checked more easily.

The layout of an element may depend on other elements that are located within it. For this reason Oryx provides the definition of a "layout" function for each stencil in the JSON file. The "layout" function is called every time the size of the element shape or the size of a child shape is recalculated. For example, the height of a pool set depends on the contained lanes within the pool set. This means, if a lane is added to the pool set, the pool set must be resized to the size of its contained lanes. The lane in a pool set does not fit into the whole pool set shape bounds, because the pool set has a shade that does not contain any elements. That is why it has to be ensured that the shade of the pool is not covered by the lane shape. As it can be seen in Listing 5.2 this can be done by using the ratio between the total pool set shape size (width: 515, height: 315) and the size of the pool set without the shade (width: 500, height: 300).

```
"layout":function(shape) {
  var lanes = shape.getChildNodes(false).findAll(
    function(node)     {
      return (node.getStencil().id() ==
           "http://b3mn.org/stencilset/bpmnplus#Lane");
    }
  );

  if(lanes.length > 0) {
    lanes = lanes.sortBy(function(lane) {
      return lane.bounds.upperLeft().y;
    });

    var shapeWidth = shape.bounds.width();
    var laneWidth = shapeWidth * 500 / 515;

    // sum up heigth of all contained lanes
    var oldShapeHeight = 0;
    lanes.each(function(lane) {
      oldShapeHeight += lane.bounds.height();
    });

    // calculate new shape height
    var newShapeHeight = oldShapeHeight * 315 / 300;
    var diff = newShapeHeight - oldShapeHeight;

    // calculate lane bounds
    var shapeHeight = diff;
    lanes.each(function(lane) {
      var ul = lane.bounds.upperLeft();
      var lr = lane.bounds.lowerRight();
      ul.y = shapeHeight;
      lr.y = ul.y + lane.bounds.height();
      shapeHeight += lane.bounds.height();
      ul.x = 30;
      lr.x = laneWidth;
      lane.bounds.set(ul, lr);
    });

    // calculate shape bounds
    var upl = shape.bounds.upperLeft();
    shape.bounds.set(upl.x, upl.y,
          shape.bounds.lowerRight().x, upl.y +
shapeHeight);
  }
}
```

Listing 5.2: "Layout" function for a pool set

Some stencil properties, which are present in the original BPMN stencil set, do not need to be entered by the modeler. They can be determined from the diagram itself (e.g. the pool attribute that defines the pool that the element is located in). That is why these elements have to be removed from the JSON definition. To determine the removed properties from the diagram, the “serialize” function of Oryx is used. This function is defined in the stencil definition like the “layout” function. As mentioned above the diagram elements and their properties get serialized to eRDF. Thus, the “serialize” function of a stencil is called every time the serialization to eRDF takes place (e.g. when the diagram is stored).

In addition to this, the “serialize” function is used to generate the mandatory ids for the elements. Unfortunately, Oryx does not provide the ability to generate the id for each element during the element creation. However, element ids must be known to the modeler for two reasons:

1. Some elements can be referenced by others using the element id (e.g. compensation intermediate event references the scope that will be compensated)
2. The transformation to BPEL4Chor uses the element ids to inform the modeler about errors in the diagram.

However, Oryx does not force the modeler to enter the ids, although the id property is denoted as a mandatory attribute in the stencil set file. That is why missing ids are generated during the serialization of the diagram. In this way the modeler only needs to define ids for elements that will be referenced by other elements or to localize transformation errors. Listing 5.3 presents the implementation of the “serialize” function. If the modeler did not enter a value for the element id, the resource id of the shape is taken.

```
"serialize":function(shape, data) {
  var id = shape.properties["oryx-id"];
  if (id == "") {
    id = shape.resourceId;
    var idRec;
    for (var i = 0; i < data.length; i++) {
      if (data[i].name == "id") {
        idRec = data[i];
        break;
      }
    }
    idRec.value=id;
  }

  return data;
}
```

Listing 5.3: “Serialize” function for a BPMN+ stencil

As mentioned above the stencil property types are restricted to “Boolean”, “Choice”, “Color”, “DateTime”, “Float”, “Integer”, “String” and “Url”. However, BPMN+ knows more complex attribute types like Correlation or Expression. Moreover, an attribute can be a list of a certain attribute type. Thus, the following ways are figured out to handle complex BPMN+ attribute types in Oryx:

- A list of string values (e.g. MessageExchanges) is entered as a whitespace separated string list. These properties still have the type “String” in the stencil set definition.

- Complex attribute types that do not occur as list (e.g. Expression) are defined for the stencil such that every child attribute of the attribute type (e.g. Expression-Language) is a property of the stencil.
- Complex attributes types (e.g. Imports) that occur in a list cannot be expressed in Oryx so far. A new property type has to be introduced for these attributes.

The latter point results in a major modification of the Oryx editor. That is why it will be explained in the following in more detail: To keep the generic definition of a stencil set, the complex property type has to be defined in the JSON file. In the editor, complex properties can be entered by a dialog that contains a table with the property values. Figure 5.2 illustrates this dialog that was implemented using the Ext library. Each column of the table is defined as "complexItem" in the stencil set definition. Complex items can again be of the type "String", "Boolean" or "Choice". Depending on this type the column in the

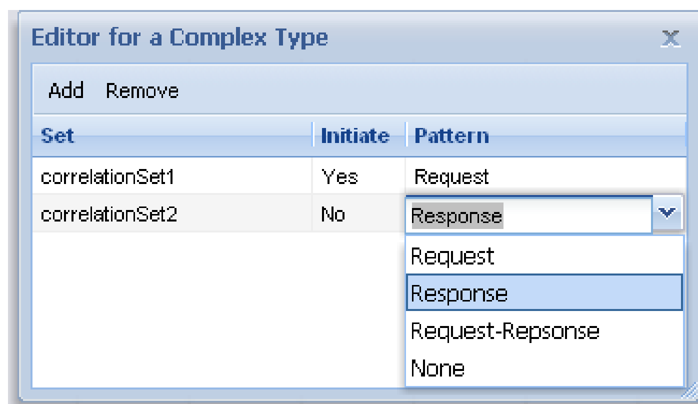


Figure 5.2: Complex property dialog for Correlations

table has an editor with a TextField, a CheckBox or a ComboBox assigned. Complex items of the type "Choice" can define the disablement of other items depending on the selected value. Imagine a complex property dialog for FromSpec properties where the FromSpecType in one row is "Opaque". In this case the other cells in the row do not need to specify a value. That is why these cells get disabled. Listing 5.4 depicts an extract of the definition of the complex property "FromSpec" in the JSON file.

The dialog for entering the complex property values is called from the property window of the editor. For this purpose the Ext library is extended with a special TriggerField that opens the dialog and holds the values from the table. To store the property values, they are serialized to the JSON format. This has the advantage that Ext already provides tables that can be filled with JSON data.

The $BPMN^+$ attribute type Property is used to define the correlation sets of a process. Each property defines a name and a type attribute. If the property has the type "Set" there can be child properties defined that represent the correlation properties. These child properties cannot be expressed using the complex property dialog since the dialog holds a list and not a tree structure. That is why the process correlation sets must be defined in an additional complex property. The complex items "name" and "properties" denote the correlation set name and properties. This is the only attribute type that cannot be expressed in Oryx as specified for $BPMN^+$.

Since there are restrictions on the containment of elements, Oryx enables the definition of containment rules for these elements. However, these rules can only define the types of elements that can be contained in other elements. The cardinality of elements or con-

```
"id":"fromspec",
"type":"Complex",
"title":"FromSpec",
"value":"",
"description":"",
"readonly":false,
"optional":true,
"complexItems": [
  {
    "id":"type",
    "name":"FromSpecType",
    "type":"Choice",
    "value":"Empty",
    "width":100,
    "optional":false,
    "items": [
      {
        "id":"c1",
        "title":"c1",
        "value":"Variable",
      },
      { ... }
    ],
    "disable": [
      {
        "value":"Variable",
        "items": [
          "property",
          "expressionlanguage",
          "expression",
          "literal"
        ]
      }, ...
    ]
  }, ... ,
  {
    "id":"property",
    "name":"Property",
    "type":"String",
    "value":"",
    "width":100,
    "optional":true,
  }, ...
]
```

Listing 5.4: Example definition of a complex property

tainment dependencies between elements cannot be considered using these rules. For this purpose Oryx defines a callback method “canContain” that is called every time an element is inserted into the diagram. In this way there can be defined more detailed restrictions on the containment. For BPMN$^+$ the “canContain” method is used for checking the following requirements:

- Timer handler contains only one timer start event
- Message handler contains only one message start event
- Fault handler, compensation handler, termination handler and scope contain only one non-triggered start event
- Lanes of one pool contain one non-triggered start event or one or more message triggered start events

- Scopes contain only one counter variable data objects and this only if the scope is a multi-instance loop

Like the containment rules, there can be defined connection rules to determine which elements can be connected with each other. In addition to the connection rules, there can be cardinality rules defined that determine how many outgoing or incoming edges of a certain type can be connected to the element. However, also in this case the rules are not sufficient for defining all restrictions of connections between elements. Many connections depend on properties that need to have certain values. Or they depend on connections to other elements like the attachment relation. Thus, the call back function "canConnect", which Oryx provides, is used to check the following requirements:

- Message flow connects activities from different swimlanes
- Sequence flow:
  - Event-based gateway is followed by intermediate message or timer events or receive tasks
  - Attached intermediate events must lead to the appropriate handler
- Directed association:
  - From participant set data object to task or scope only if they are multi-instance loops
  - From error event to fault variable only if error event is attached
  - From compensation intermediate event to compensation handler only if compensation event is attached and there is no other compensation handler connected with the event
  - From message start event to message variable data object only if message start event is located within a message handler
  - From message start event to standard variable data object only if message start event is not located within a message handler
  - Association from standard variable data object to error event only if error event is not attached (throw not catch)

These restrictions should help the modeler to create diagrams that can be transformed to BPEL4Chor.

A validation that involves the element attributes has to be implemented in a plugin. There is no mechanism like rules or call back functions to ease this validation, because it has to be done after the elements were inserted into the model. An exhaustive validation would lead to a bulk of JavaScript code and this would be difficult to maintain. Since the validation is also done in the transformation, it would be a better way to enhance the error messages of the transformation and processes them in the editor. The error message from the transformation could be stored as property for the incorrect element. Moreover, the incorrect element could be highlighted in the model. In this way a double validation in the editor and in the transformation can be avoided.

5.3 Calling the Transformation Web Service

To transform BPMN$^+$ diagrams to BPEL4Chor, the transformation web service must be called from the Oryx editor. For this purpose a new plugin is created and added to the toolbar of the editor as shown in Figure 5.3. This plugin must prepare the input data, call the transformation and present the results.

Figure 5.3: Transformation plugin in the toolbar

Every plugin class must offer its functionality in the constructor. In this way properties like the name or the icon can be determined. Listing 5.5 depicts the offering of the plugin for the BPEL4Chor transformation. The “functionality” property identifies the function “transform” to implement the plugin functionality.

```
this.facade.offer({
  'name':"ExportBPEL4Chor",
  'functionality': this.transform.bind(this),
  'group': "BPEL4Chor",
  'icon': ORYX.PATH + "images/export.png",
  'description': "Export diagram to BPEL4Chor",
  'index': 1,
  'minShape': 0,
  'maxShape': 0});
```

Listing 5.5: Offering the plugin functionality

First of all, the function “transform” needs to provide the input data for the transformation web service. The web service takes XPDL4Chor (see chapter 2.4) as input. So XPDL4Chor needs to be generated from the BPMN$^+$ diagram. As mentioned above the diagram data is stored in eRDF. The eRDF is embedded in XHTML, which is an XML format. XPDL4Chor is an XML format, too, so the Extensible Stylesheet Language Transformation (XSLT, [XSLT99]) can be used to transform the eRDF data to XPDL4Chor.

The data of a diagram element is located in `span` elements contained in the `div` element that represents the diagram element. Each `span` element has a class attribute to identify the property value that the `span` contains. The property value itself is the text content of the `span` element. An example for the serialized properties of a pool set is depicted in Listing 5.6.

```
<div id="resource0">
    <span class="oryx-type">
        http://b3mn.org/stencilset/bpmnplus#PoolSet
    </span>
    <span class="oryx-poolsetid">PoolSetId</span>
    ...
    <span class="oryx-messageexchanges">
        exchange1 exchange2
    </span>
    ...
</div>
```

Listing 5.6: Excerpt from serialized pool set properties

XSLT is a language that describes rules for transforming a source XML tree into a result XML tree. The transformation is expressed using template rules. If a pattern matches against elements in the source tree, the template is used to create a part of the result tree. Listing 5.7 presents an example XSLT template that splits a whitespace separated string

list to generate an XPDL4Chor `MessageExchange` element for each resulting value. This template is called for the “oryx-messageexchanges” property that can be seen in Listing 5.6. In addition to this example template, templates have been created for generating the XPDL4Chor representations of the diagram elements that are needed for the transformation to BPEL4Chor.

```
<xsl:template name="selects-tokens">
   <xsl:param name="list" />
   <xsl:variable name="newlist"
      select="concat(normalize-space($list), ' ')" />
   <xsl:variable name="first"
      select="substring-before($newlist, ' ')" />
   <xsl:variable name="remaining"
      select="substring-after($newlist, ' ')" />
   <xsl:if test="string-length(normalize-space($first))>0">
      <chor:Selects>
         <xsl:value-of select="$first" />
      </chor:Selects>
   </xsl:if>
   <xsl:if test="$remaining">
      <xsl:call-template name="selects-tokens">
         <xsl:with-param name="list" select="$remaining" />
      </xsl:call-template>
   </xsl:if>
</xsl:template>
```

Listing 5.7: XSLT template for generating the XPDL4Chor selects element

Firefox[1] is the target browser of Oryx. It already comes with an XSLT support – an XSLT processor that can be accessed by JavaScript. The class that provides this functionality is called “XSLTProcessor”. Listing 5.8 shows the usage of the “XSLTProcessor” class in the transformation plugin. First the XSLT stylesheet is loaded to the document “xslRef”. When the stylesheet is loaded, it is imported to the XSLT processor using the function “importStylesheet”. After that the diagram data is serialized using the “DataManager” class of Oryx. The serialized data does not have a root node, because it only creates the `div` element for each graphical object. That is why a root node (“data”) has to be added to the document. Only with this root node the serialized string can be parsed to a DOM document using the DOMParser. Now this document can be transformed to XPDL4Chor with the XSLT processor using the function “transformToDocument”.

The Oryx editor can define any element of a stencil set as the canvas element. The canvas element is the top-level parent container of all other elements that will be inserted into the model. To transform a model to BPEL4Chor, the “BPMN+Diagram” element has to be the canvas element. The DataManager class does not serialize the canvas element and its properties. So canvas properties like the diagram name or the target namespace have to be added after the XSLT transformation. This is done in the function “addCanvasProperties”.

Now that the input data of the transformation is available, the web service can be called. For this purpose Firefox provides the “SOAPCall” class, which is used to initiate and in-

1 See http://www.mozilla.com/firefox/

```
var xsltProcessor = new XSLTProcessor();
var xslRef =
  document.implementation.createDocument("", "", null);
var index = location.href.lastIndexOf("/");
var xslt = location.href.substring(0, index) +
"/xslt/BPMNplus2XPDL4Chor.xsl";
xslRef.load(xslt);

xslRef.onload = function() {
  xsltProcessor.importStylesheet(xslRef);

  var serializedDOM = DataManager.serializeDOM(this.facade);
  serializedDOM = "<data>" + serializedDOM + "</data>";

  var parser=new DOMParser();
  var doc=parser.parseFromString(serializedDOM,"text/xml");

  var xpdl4Chor = xsltProcessor.transformToDocument(doc);
  this.addCanvasProperties(xpdl4Chor);
  ...
}
```

Listing 5.8: XSLT processing with JavaScript

voke a SOAP[1] call. Listing 5.9 shows the call of the transformation web service using the "SOAPCall" class. First of all the XPDL4Chor document needs to be serialized, since the web service expects the diagram as string parameter. After that, the call is initialized with the transport URI that is defined in the configuration file of the editor. This matches the web service location defined in the WSDL file of the web service. The parameters passed with the SOAP message are generated as name/value pairs using the "SOAPParameter" class. Then the message gets encoded with the requested method and the provided parameters using the "encode" function. Finally, the SOAP call is invoked asynchronous by the function "asyncInvoke".

```
var serialized = (new XMLSerializer()).
  serializeToString(xpdl4Chor);

var call = new SOAPCall();
call.transportURI = ORYX.CONFIG.TRANS_URI;
call.actionURI = "";

var inputDiagram = new SOAPParameter();
inputDiagram.name = "diagramStr";
inputDiagram.value = serialized;

var inputValidate = new SOAPParameter();
inputValidate.name= "validate";
inputValidate.value = true;

call.encode(0, "transform",
  ORYX.CONFIG.TRANS_SERVER, 0, null, 2,
  new Array(inputDiagram,inputValidate));

call.asyncInvoke(this.result.bind(this));
```

Listing 5.9: SOAP call with JavaScript

1 See http://www.w3.org/TR/soap/

JavaScript code runs under the restriction of the Same Origin Policy[1] in the Firefox Browser. That is why a web service can only be called if the protocol, host and port are equal to the calling page. If the web service runs on a different server, the only way to by-pass this restriction is to digitally sign the calling script. To sign a script, it must be located in a jar file and this jar file can be signed using certain sign tools[2]. Since this would be a major change to the Oryx editor, the signing of the editor is left to the future development. Thus, the current implementation assumes that the web service runs on the same server as the editor.

The result of the transformation is presented to the modeler in a dialog. The dialog is implemented using the Ext library. If an error occurred during the parsing of the diagram (e.g. because of missing attribute values), the dialog just shows a text area containing the error message. If the transformation returns a result, the topology and the processes are presented in a table. The table shows if the transformation was successful or if the XML could not be generated for the topology or a process. Moreover, the dialog provides the options to show the result in a browser window, to save a single result file as XML or to save all results as ZIP file in the local file system. Figure 5.4 illustrates the result dialog, where the topology was not generated correctly.

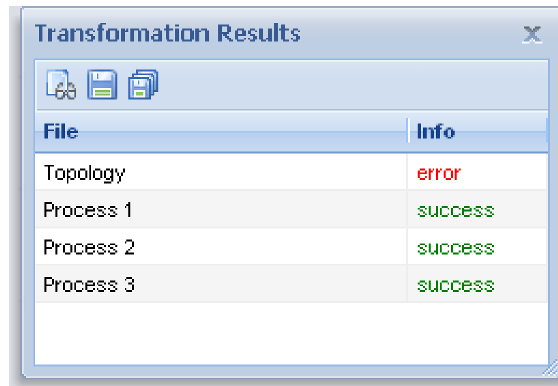


Figure 5.4: Result dialog for the transformation

To conclude this chapter, Figure 5.5 illustrates how the extensions fit into the editor architecture. The illustration uses the FMC notation (see [KGT06]).

1 See http://www.mozilla.org/projects/security/components/same-origin.html
2 See http://www.mozilla.org/projects/security/components/terms.html#cert

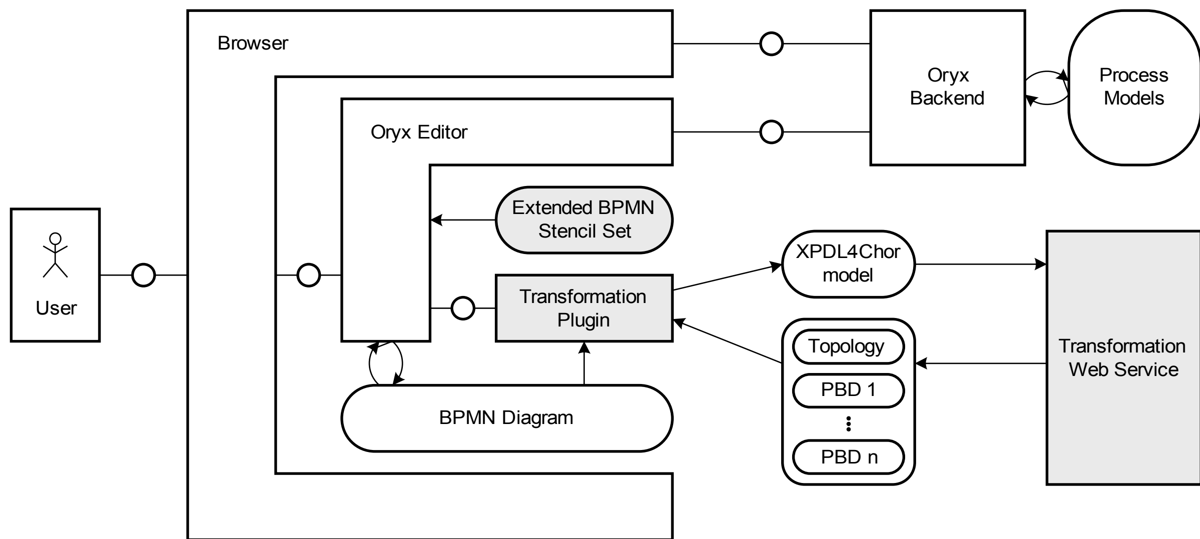


Figure 5.5: Extensions in the Oryx architecture

6 Summary and Outlook

This thesis has brought together the graphical notation BPMN with the XML-based choreography language BPEL4Chor. It has been analyzed to what extent BPMN can be used for modeling BPEL4Chor choreographies. The deficiencies of BPMN in this case have motivated the need of a BPMN extension ($BPMN^+$) for expressing the complete BPEL4Chor element set. In addition to this extension, a transformation was introduced to generate the BPEL4Chor choreography from a $BPMN^+$ diagram. Finally, $BPMN^+$ and the transformation were integrated in the Oryx editor to provide a graphical development of BPEL4Chor choreographies using $BPMN^+$.

The notation and transformation provided in this thesis can be used in the whole development life cycle of choreographies and the resulting processes. Business process choreographies mainly describe the communication between processes. Thus, they only need to define communicating activities. Details about the implementation of the participating processes can be left open. Nevertheless, a choreography can be the starting point for the implementation of more detailed processes in top-down development approaches. Based on the choreography, every process can be refined without knowing details about the other processes. For this purpose, $BPMN^+$ provides the ability to model non-communicating activities as well. In this way complete processes can be modeled based on the choreography. To generate the final executable processes, there is no need to import the choreography or the abstract processes into other tools. They can be generated from BPEL4Chor using the transformations presented in [Reim07]. In future work this transformation may be implemented and integrated into the Oryx editor as well.

Since the Oryx editor is still under development, there may be implemented some improvements in the future. First of all a verification after the diagram was modeled should be realized. In this way also the element attributes can be taken into account for the verification. Moreover, a better verification mechanism could ensure that there can be modeled only diagrams that are transformable to BPEL4Chor. For instance there may be implemented a check for the soundness and safeness of the diagram and that the diagram does not contain any arbitrary cycles.

7 Bibliography

(All links in this thesis were last followed on November 28th, 2007)

[BDtH05] Alistair Barros, Marlon Dumas and Arthur ter Hofstede. Service Interaction Patterns. In Aalst, W., Benatallah, B., Casati, F., eds.: Business Process Management, volume 3649 of LNCS, Berlin, Springer Verlag, pages 302–318, 2005

[BFV07] Giso Bartels, Philipp Frank, Marco Völz. Vergleich von BPMN-Modellierungswerkzeugen. Fachstudie Nr. 75, IAAS, Universität Stuttgart, Juli 2007

[BPEL07] Web Services Business Process Execution Language Version 2.0. OASIS Standard, April 2007.

[BPMN06] Business Process Modeling Notation (BPMN) Specification. Final Adopted Specification, Object Management Group (OMG), February 2006.

[BPMN07] Business Process Modeling Notation (BPMN) 2.0 Request for Proposal. Object Management Group (OMG), June 2007.

[CLS+05] Francisco Curbera, Frank Leymann, Tony Storey, Donald F. Ferguson, and Sanjiva Weerawarana. Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WSBPEL, WS-Reliable Messaging and More. Prentice Hall PTR, 2005.

[Czuc07] Martin Czuchr. Oryx: Embedding Business Process Data into the Web. Final Bachelor's Paper, Hasso Plattner Institut, University Potsdam, June 2007

[DDO07] Remco M. Dijkman, Marlon Dumas and Chun Ouyang. Formal Semantics and Automated Analysis of BPMN Process Models. Science of Computer Programming, 67(2-3):162–198, July 2007

[Deck06] Gero Decker. Process Choreographies in Service-oriented Environments. Master Thesis, Hasso-Plattner-Institute, University of Potsdam, Germany, October 2006

[DePu07] Gero Decker and Frank Puhlmann. Extending BPMN for Modeling Complex Choreographies. Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS), Vilamoura, Portugal, November 2007

[DKLW07] Gero Decker, Oliver Kopp, Frank Leymann, Mathias Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In Proceedings International Conference on Web Services (ICWS), July 2007

[KBR+05] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon, Charlton Barreto. Web Services Choreography Description Language Version 1.0, W3C Candiate Recommendation. Technical Report, November 2005. http://www.w3.org/TR/ws-cdl-10

[KGT06] Andreas Knöpfel, Bernhard Gröne, Peter Tabeling. Fundamental Modeling Concepts: Effective Communication of IT Systems. Wiley, March 2006, ISBN 978-0-470-02710-3

[LA94] Frank Leymann and Wolfgang Altenhuber. Managing business processes as an information resource. IBM Systems Journal, 33(2):326–348, 1994.

[LSW97] Peter Langner, Christoph Schneider, Joachim Wehler. Ereignisgesteuerte Prozeßketten und Petri-Netze. Report Series of the Department of Computer Science 196, University of Hamburg - Computer Sience Department, Germany, March 1997

[MLZ05] Jan Mendling, Kristian Bisgaard Lassen, Uwe Zdun. Transformation Strategies between Block-Oriented and Graph-Oriented Process Modelling Languages. Technical Report JM-200510-10, WU Vienna, October 2005

[ODtHvdA07] Chun Ouyang, Marlon Dumas, Arthur H.M. ter Hofstede and Wil M.P. van der Aalst. Pattern-based Translation of BPMN Process Models to BPEL Web Services. International Journal of Web Services Research (JWSR), March 2007

[ODvdAtH06] Chun Ouyang, Marlon Dumas, Wil M.P. van der Aalst and Arthur H.M. ter Hofstede. From Business Process Models to Process-oriented Software Systems: The BPMN to BPEL Way. BPM Center Report BPM-06-26, October 2006.

[Palm06] Nathaniel Palmer. Understanding the BPMN-XPDL-BPEL Value Chain. Business Integration Journal, pages 54 – 55, November/December 2006

[PDKL07] Kerstin Pfitzner, Gero Decker, Oliver Kopp, Frank Leymann. Web Service Choreography Configurations for BPMN. Proceedings of the 3rd International Workshop on Engineering Service-oriented Applications: Analysis, Design and Composition (WESOA), Vienna, Austria, September 2007

[Reim07] Peter Reimann. Generating BPEL Processes from a BPEL4Chor Description. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Infor-

mationstechnik, Studienarbeit 2100, September 2007.

[ReMe06] Jan C. Recker and Jan Mendling. On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. 11th International Workshop on Exploring Modeling Methods in Systems Analysis and Design, June 2006

[RtHvdAM06] Nick Russell, Arthur H.M. ter Hofstede, Wil M.P. van der Aalst and Nataliya Mulyar. Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22, BPMcenter.org, 2006

[SVG03] Scalable Vector Graphics (SVG) 1.1 Specification. W3C Recommendation, January 2003.

[Tsch07] Willi Tscheschner. Oryx Dokumentation. Final Bachelor's Paper, Hasso Plattner Institut, University Potsdam, 2007

[vdAa97] Wil M.P. van der Aalst. Verification of Workflow Nets. Proceedings of the 18th International Conference on Application and Theory of Petri Nets (ICATPN), Springer Verlag, pages 407 - 426, June 1997

[Wesk07] Mathias Weske. Business Process Management Concepts, Languages Architectures. Springer-Verlag, pages 227 – 257, 2007, ISBN 978-3540735212

[WEvdA+05] Moe Thandar Wynn, David Edmond, Wil M.P. van der Aalst and Arthur H.M. ter Hofstede. Achieving a General, Formal and Decidable Approach to the OR-join in Workflow using Reset nets. Proceedings 26th International Conference on Applications and Theory of Petri Nets (ICATPN) LNCS 3536, pages 423-443, 2005

[Whi05] Stephen A. White. Using BPMN to Model a BPEL Process. BPTrends, 3(3):1-18, March 2005

[WvdA+02] Petia Wohed, Wil M.P. van der Aalst, Marlon Dumas and Arthur H.M. ter Hofstede. Pattern Based Analysis of BPEL4WS. QUT Technical Report, FIT-TR-2002-04, Faculty of Information Technology, Queensland University of Technology, December 2002. http://www.workflowpatterns.com/documentation/documents/qut_bpel_rep.pdf

[XPath99] XML Path Language (XPath) Version 1.0. W3C Recommendation, November 1999.

[XPDL05] Process Definition Interface -- XML Process Definition Language, Version 2.0. Workflow Management Coalation Workflow Standard, October 2005.

[XSLT99] XSL Transformation (XSLT) Version 1.0. W3C Recommendation, November 1999.

[ZBDtH06] Johannes M. Zaha, Alistair Barros, Marlon Dumas and Arthur ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. The Fourteenth International Conference on Cooperative Information Systems (CoopIS), Montpellier, France, 2006, LNCS

A BPMN$^+$ Element Attributes

The following tables list the complete set of BPMN$^+$ element attributes. These are the attributes from the BPMN elements and the attributes introduced in BPMN$^+$.

The last column of each table indicates whether an attribute is unchanged, adapted or added to the list of attributes defined in BPMN. The attributes are marked as follows:

- Attribute introduced in BPMN$^+$: [+]
- Attribute adapted in BPMN$^+$: [*]
- Attribute used unchanged: [ ]

There is a special notation for defining the type of an attribute. Some attributes can have alternative values, which are separated by "|" and grouped within "(" and ")". To specify the default value of an attribute, it is separated from the actual type using a colon.

Notation for a type declaration: (alternative 1 | ... | alternative n) alternative i : type

A.1 Business Process Diagram

The following table displays the set of attributes for a business process diagram (BPD).

Attribute Name	Type	Description	
Id	Object	See table 8.6 in [BPMN06]	[ ]
Name	String	See table 8.6 in [BPMN06]	[ ]
Version (0-1)	String	See table 8.6 in [BPMN06]	[ ]
Author (0-1)	String	See table 8.6 in [BPMN06]	[ ]
Language (0-1)	English : String	See table 8.6 in [BPMN06]	[ ]
ExpressionLanguage (0-1)	"urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0" : String	See table 8.6 in [BPMN06] In BPMN$^+$, the default language is XPath.	[*]
QueryLanguage (0-1)	"urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0" : String	See table 8.6 in [BPMN06] In BPMN$^+$, the default language is XPath.	[*]
CreationDate (0-1)	Date	See table 8.6 in [BPMN06]	[ ]
ModificationDate (0-1)	Date	See table 8.6 in [BPMN06]	[ ]
Pools (0-n)	Pool	See table 8.6 in [BPMN06] At least one pool or pool set has to be spe-	[*]

		cified.	
PoolSets (0-n)	Pool Set	A BPD can contain pool sets. At least one pool or pool set has to be specified.	[+]
Documentation (0-1)	String	See table 8.6 in [BPMN06]	[ ]
TargetNamespace (0-1)	"urn:choreography/topology": String	This is the target namespace used for the participant topology.	[+]
GroundingFile (0-1)	String	Since the grounding will not be modeled within a BPMN+ diagram, a grounding file can be associated with the choreography.	[+]

A.2 Process

The following table displays the set of attributes for a process.

Attribute Name	Type	Description	
Id	Object	See table 8.7 in [BPMN06]	[ ]
Name	String	See table 8.7 in [BPMN06]	[ ]
ProcessType	Abstract : String	See table 8.7 in [BPMN06] BPEL4Chor uses abstract BPEL processes. Thus, the process type must be Abstract if the diagram will be transformed to BPEL4Chor.	[*]
Status	(None | Ready | Active | Cancelled | Aborting | Aborted | Completing | Completed) None : String	See table 8.7 in [BPMN06]	[ ]
GraphicalElements (0-n)	Object	See table 8.7 in [BPMN06]	[ ]
Assignments (0-n)	Assignment	See table 8.7 in [BPMN06]	[ ]
Properties (0-n)	Property	See table 8.7 in [BPMN06]	[ ]
AdHoc	False : Boolean	See table 8.7 in [BPMN06] BPMN+ does not support ad hoc processes, so the value of this attribute is always False.	[*]
SuppressJoinFailure (0-1)	False : Boolean	See table 8.7 in [BPMN06]	[ ]

EnableInstanceCom-pensation	False : Boolean	See table 8.7 in [BPMN06] This attribute always has the value False in BPMN+.	[*]
Categories (0-n)	String	See table 8.7 in [BPMN06]	[ ]
Documentation (0-1)	String	See table 8.7 in [BPMN06]	[ ]
QueryLanguage (0-1)	String	This attribute value overwrites the QueryLanguage attribute value defined for the BPD.	[+]
Expression Language (0-1)	String	This attribute value overwrites the ExpressionLanguage attribute value defined for the BPD.	[+]
ExitOnStandard-Fault (0-1)	False : Boolean	If set to False, the process can handle standard faults using fault handlers. If set to True, the process must exit immediately when a standard fault is encountered.	[+]
MessageExchanges (0-n)	String	This attribute defines message exchange names that can be used to disambiguate the relationship between inbound communicating flow objects and send tasks.	[+]

A.3 Graphical Object

The following table displays the set of common attributes for a graphical object.

Attribute Name	Type	Description	
Id	Object	See table 9.1 in [BPMN06]	[ ]
Categories (0-n)	String	See table 9.1 in [BPMN06]	[ ]
Documentation (0-1)	String	See table 9.1 in [BPMN06]	[ ]

A.4 Swimlane

The following table displays the set of common attributes for swimlanes, which extends the set of common graphical objects attributes.

Attribute Name	Type	Description	
Name	String	See table 9.31 in [BPMN06]	[ ]

A.4.1 Pool

The following table displays the set of attributes for a pool, which extends the set of common swimlane attributes.

Attribute Name	Type	Description	

Process	Process	See table 9.33 in [BPMN06] In BPMN+, each pool must contain a process.	[*]
Participant	Participant	See table 9.33 in [BPMN06] The name of the Participant is the name of the participant reference the pool represents.	[*]
Selects (0-n)	String	This attribute defines the names of the participants that are selected by the participant reference this pool represents.	[+]
Containment (0-1)	(Required | MustAdd | AddIfNotExists) AddIfNotExists : String	This attribute describes the containment relationship of the participant reference represented by this pool in a participant set. This attribute must only be set if the participant reference is contained in a participant set.	[+]
Lanes (1-n)	Lane	See table 9.33 in [BPMN06]	[ ]
BoundaryVisible	True : Boolean	See table 9.33 in [BPMN06]	[ ]
TargetNamespace	"urn:choreography/process" : String	This is the target namespace of the top-level process in the pool.	[+]
Prefix	"p": String	The prefix belongs to the target namespace above. It is used for referencing the process and its elements that are located in the pool.	[+]
Imports (0-n)	Import	The import elements declare dependencies on external XML schema or WSDL definitions.	[+]

A.4.2 Pool Set

The following table displays the set of attributes for a pool set, which extends the set of common swimlane attributes.

Attribute Name	Type	Description	
Process	Process	This attribute defines the top-level process that is contained within the pool set. In BPMN+, each pool set must contain a process.	[+]
Lanes (1-n)	Lane	Lanes do not add any semantical value in BPMN+, but they can be used for layout purposes.	[+]
BoundaryVisible	True : Boolean	This attribute defines whether the boundary of the pool set is visible in the diagram.	[+]
TargetNamespace	"urn:choreogra-	This is the target namespace of the process	[+]

	phy/process” : String	contained in the pool set.	
Prefix	“p”: String	The prefix belongs to the target namespace above and it is used for referencing the process and its elements that are located in the pool set.	[+]
Imports (0-n)	Import	The import elements declare dependencies on external XML Schema or WSDL definitions.	[+]

A.4.3 Lane

The following table displays the set of attributes for a lane, which extends the set of common swimlane attributes.

Attribute Name	Type	Description	
ParentPool (0-1)	Pool	See table 9.34 in [BPMN06] It must be specified either a parent pool or a parent pool set.	[*]
ParentPoolSet (0-1)	PoolSet	This is the parent pool set of the lane. It must be specified either a parent pool or a parent pool set.	[+]
ParentLane (0-1)	Lane	See table 9.34 in [BPMN06]	[ ]

A.5 Artifact

The following table displays the common set of attributes for an artifact, which extends the set of graphical object attributes.

Attribute Name	Type	Description	
ArtifactType	(DataObject | Group | Annotation) DataObject : String	See table 9.35 in [BPMN06]	[ ]
Pool (0-1)	Pool	See table 9.35 in [BPMN06] An artifact is either located in a pool, in a pool set or outside of those.	[*]
PoolSet (0-1)	Pool Set	An artifact is either located in a pool, in a pool set or outside of those. This attribute specifies the pool set, the artifact is located in.	[+]
Lanes (0-n)	Lane	See table 9.35 in [BPMN06]	[ ]

A.5.1 Annotation

The following table displays the set of attributes for an annotation, which extends the set of artifact attributes.

Attribute Name	Type	Description	
Text	String	See table 9.37 in [BPMN06]	[ ]

A.5.2 Group

The following table displays the set of attributes for a group, which extends the set of artifact attributes.

Attribute Name	Type	Description	
Name (0-1)	String	See table 9.38 in [BPMN06]	[ ]

A.5.3 Data Object

The following table displays the set of attributes for a data object, which extends the set of artifact attributes.

Attribute Name	Type	Description	
Name	String	See table 9.36 in [BPMN06]	[ ]
State (0-1)	String	See table 9.36 in [BPMN06]	[ ]
Properties (0-n)	Properties	See table 9.36 in [BPMN06]	[ ]
RequiredForStart	False : Boolean	See table 9.36 in [BPMN06] This attribute is always False in BPMN+, since data objects do not block the start of an activity.	[*]
ProducedAtCompletion	False : Boolean	See table 9.36 in [BPMN06] The attribute value depends on the data object type in BPMN+. A variable data object that is associated with an activity as output variable has this attribute set to True. For other data objects this attribute is set to False.	[*]
Type	(Variable | Reference | Set) : String	A data object in BPMN+ can either represent a variable, a participant reference or a participant set.	[+]

Variable Data Object

Attribute Name	Type	Description	
VariableType	(Counter | Fault | Message |	BPMN+ introduces different data object types. A standard variable expresses the de-	[+]

	Standard) Standard : String	fined variables within a process or sub-process. The counter variable is used as counter within a multi-instance loop. The fault variable holds the data of a fault that was caught and the message variable contains the message that triggers a message event handler.	
[VariableType = Standard] Scope (0-1)	Scope	A standard variable data object can be limited to a specified scope.	[+]
[VariableType = Standard, Message or Fault] DefinitionType	(MessageType | XMLType | XMLElement) : String	Variables can be defined in terms of WSDL message types, XML schema types or XML schema elements. If the VariableType is set to Fault or Message, this attribute must have the value MessageType or XMLElement.	[+]
[VariableType = Standard, Message or Fault] DefinitionTypePrefix	String	The prefix identifies the namespace of the imported file, the definition type of the variable data object is located in.	[+]
[VariableType = Standard, Message or Fault] DefinitionTypeValue	String	This attribute value has to match a WSDL message type, XML schema type or XML schema element defined in an imported file. If the VariableType is set to Fault, an XML schema type is not allowed.	[+]
[VariableType = Standard] VariableFromSpec (0-1)	FromSpec	With the FromSpec a defined variable can be initialized. The attributes of a FromSpec can be found in section FromSpec.	[+]

Participant Reference Data Object

Attribute Name	Type	Description	
Selects (0-n)	String	This attribute defines the names of the participants that are selected by this participant reference.	[+]
Containment (0-1)	(Required | MustAdd | AddIfNotExists) AddIfNotExists : String	This attribute describes the containment relationship of the participant reference in a participant set. This attribute must only be set if the participant reference is contained in a participant set.	[+]
CopyTo (0-1)	String	If the participant reference data object is associated with a message flow the reference can be copied to another reference.	[+]

		The name of this reference is specified in the CopyTo attribute.	
Scope (0-1)	Scope	The Scope attribute defines the scope the participant reference is limited to.	[+]

Participant Set Data Object

Attribute Name	Type	Description	
CopyTo (0-1)	String	If the participant set data object is associated with a message flow the set can be copied to another set. The name of this set is specified in the CopyTo attribute.	[+]
Scope (0-1)	Scope	The Scope attribute defines the scope the participant set is limited to.	[+]

A.6 Connecting Object

The following table displays the common set of attributes for a connecting object which extends the set of graphical object attributes.

Attribute Name	Type	Description	
Name	String	See table 10.1 in [BPMN06]	[ ]

A.6.1 Sequence Flow

The following table displays the set of attributes for a sequence flow which extends the set of connecting object attributes.

Attribute Name	Type	Description	
Source	Flow Object	See table 10.1 in [BPMN06] An end event cannot be the source of a sequence flow.	[*]
Target	Flow Object	See table 10.1 in [BPMN06] A start event cannot be the target of a sequence flow.	[*]
ConditionType	(None | Expression | Default) None : String	See table 10.2 in [BPMN06]	[ ]
[ConditionType = Expression] ConditionExpression	Expression	See table 10.2 in [BPMN06]	[ ]
Quantity	1: Integer	See table 10.2 in [BPMN06] In BPMN$^+$, this attribute is always 1.	[*]

A.6.2 Message Flow

The following table displays the set of attributes for a message flow which extends the set of connecting object attributes.

Attribute Name	Type	Description	
Source	Task	See table 10.1 in [BPMN06] This object can only be a service or a send task.	[*]
Target	Flow Object	See table 10.1 in [BPMN06] This activity can only be a service task, a receive task, a message start or a message intermediate event.	[*]
Message	Message	See table 10.3 in [BPMN06] In BPMN+, this attribute is mandatory.	[*]

A.6.3 Association

The following table displays the set of attributes for an Association which extends the set of Connecting Object attributes.

Attribute Name	Type	Description	
Direction	(None | To | From) None: String	See table 10.4 in [BPMN06] In BPMN+, the direction Both is not used.	[*]
Source	Object	See table 10.1 in [BPMN06] If the direction is None, the source has to be a message flow. If the direction is From, it has to be a message start event, a message intermediate event, an attached error intermediate event, a service task or an attached compensation event. If the direction is To, it has to be a send task, a service task or a compensation handler.	[*]
Target	Data Object	See table 10.1 in [BPMN06] If the source is an attached compensation event, the target must be a compensation handler. If the source is a compensation handler, the target must be an attached compensation event.	[*]

A.7 Flow Object

The following table displays the set of common attributes for flow objects, which extends the set of common graphical object attributes.

Attribute Name	Type	Description	
Name	String	See table 9.2 in [BPMN06]	[ ]

Assignments (0-n)	Assignment	See table 9.2 in [BPMN06]	[ ]
Pool (0-1)	Pool	See table 9.2 in [BPMN06] Each flow object is located either in a pool or in a pool set.	[*]
PoolSet (0-1)	Pool Set	This attribute identifies the pool set, the flow object belongs to. Each flow object is located either in a pool or in a pool set.	[+]
Lanes (0-n)	Lane	See table 9.2 in [BPMN06]	[ ]

A.7.1 Gateway

The following table displays the set of common attributes for a gateway which extends the set of common flow object attributes.

Attribute Name	Type	Description	
GatewayType	(XOR | AND | OR) XOR : String	See table 9.25 in [BPMN06] In BPMN+, only exclusive, parallel and inclusive gateways are used.	[*]

Parallel Gateway

Attribute Name	Type	Description	
Gates (1-n)	Gate	See table 9.30 in [BPMN06] In BPMN+, a parallel gateway must have at least one outgoing gate.	[*]

Exclusive Gateway

Attribute Name	Type	Description	
XORType	(Data | Event) Data : String	See table 9.26 in [BPMN06]	[ ]
MarkerVisible	False : Boolean	See table 9.26 in [BPMN06]	[ ]
[XORType = Data] Gates (1-n)	Gate	See table 9.26 in [BPMN06] In BPMN+, an exclusive gateway must have at least one outgoing gate.	[*]
[XORType = Data] DefaultGate (0-1)	Gate	See table 9.26 in [BPMN06]	[ ]
[XORType = Event] Gates (2-n)	Gate	See table 9.27 in [BPMN06]	[ ]

[XORType = Event] Instantiate	Boolean	See table 9.27 in [BPMN06]	[ ]

Inclusive Gateway

Attribute Name	Type	Description	
Gates (1-n)	Gate	See table 9.28 in [BPMN06] In BPMN+, an inclusive gateway must have at least one outgoing gate.	[*]
DefaultGate (0-1)	Gate	See table 9.28 in [BPMN06]	[ ]

A.7.2 Event

The following table displays the set of common attributes for an event which extends the set of flow object attributes.

Attribute Name	Type	Description	
EventType	(Start | End | Intermediate) Start : String	See table 9.3 in [BPMN06]	[ ]

Start Event

Attribute Name	Type	Description	
Trigger	(None | Message | Timer) None : String	See table 9.5 in [BPMN06] In BPMN+, only start events with the trigger None, Message and Timer are used.	[*]
[Trigger = Message] Message	Message	See table 9.5 in [BPMN06] In BPMN+, the message is already provided by the message flow connected to the start event.	[*]
[Trigger = Message] Implementation	(Web Service | Other | Unspecified) Web Service : String	See table 9.5 in [BPMN06]	[ ]
[Trigger = Message] MessageExchange (0-1)	String	This attribute defines a message exchange name that can be used to disambiguate the relationship between this event and the following send tasks.	[+]
[Trigger = Message] Correlations (0-n)	Correlation	The Correlations attribute is used to identify a conversation. The attributes of a correlation can be found in 2.3.21 Supporting Types.	[+]
[Trigger = Message] OpaqueOutput	True: Boolean	The output variable can be opaque to hide the actual data that is received. If this attri-	[+]

		bute is True, the FromParts must not be specified and there must not be an output variable associated with this event.	
[Trigger = Message] FromParts (0-n)	FromPart	The FromParts can be used instead of modeling an association to the data object containing the received Message.	[+]
[Trigger = Timer] TimeType	(Date | Cycle) Date : String	This attribute is necessary to distinguish opaque from non-used time definitions.	[+]
[TimeType = Date] TimeDate (0-1)	Date	See table 9.5 in [BPMN06]	[ ]
[TimeType = Cycle] TimeCycle (0-1)	String	See table 9.5 in [BPMN06]	[ ]
[Trigger = Timer] TimeLanguage (0-1)	String	Defines the language that is used to express the TimeCycle or the TimeDate value.	[+]
[Trigger = Timer] RepeatEvery (0-1)	Expression	The RepeatEvery attribute holds a duration expression that defines the interval in which the event will be triggered repeatedly.	[+]

End Event

Attribute Name	Type	Description	
Result	None: String	See table 9.7 in [BPMN06] There are only end events with a result of None in BPMN$^+$.	[*]

Intermediate Event

Attribute Name	Type	Description	
Trigger	(Message | Timer | Error | Compensation | Termination) Message : String	See table 9.9 in [BPMN06] In BPMN$^+$, only intermediate events with the trigger Message, Timer, Error, Compensation and Termination are used.	[*]
SuppressJoinFailure	False: Boolean	This attribute determines if join failures should be suppressed when there are multiple paths that lead to the event.	[+]
[Trigger = Error, Compensation or Termination]	Object	See table 9.9 in [BPMN06] In BPMN$^+$, only intermediate events with the trigger Error, Compensation and Ter-	[*]

Target (0-1)		mination can be attached.	
[Trigger = Message] Message	Message	See table 9.9 in [BPMN06] In BPMN+, the message is already provided by the message flow connected to the event.	[*]
[Trigger = Message] Implementation	(WebService | Other | Unspecified) WebService : String	See table 9.9 in [BPMN06]	[ ]
[Trigger = Message] MessageExchange (0-1)	String	This attribute defines a message exchange name that can be used to disambiguate the relationship between the message event and the following send tasks.	[+]
[Trigger = Message] FromParts (0-n)	FromPart	If there is an incoming message flow connected with to event, the FromParts can be set. This is used instead of modeling an association to the variable data object containing the received message data.	[+]
[Trigger = Message] OpaqueOutput	True: Boolean	The output variable can be opaque to hide the actual data that is received. If this attribute is True, the FromParts must not be specified and there must not be an output variable associated with this event.	[+]
[Trigger = Message] Correlations (0-n)	Correlation	The Correlations attribute is used to identify a conversation. The attributes for a Correlation can be found in 2.3.21 Supporting Types.	[+]
[Trigger = Timer] TimeType	(Date | Cycle) Date : String	This attribute is necessary to distinguish opaque from non-used time expressions.	[+]
[Trigger = Timer] TimeDate (0-1)	Date	See table 9.9 in [BPMN06]	[ ]
[Trigger = Timer] TimeCycle (0-1)	String	See table 9.9 in [BPMN06]	[ ]
[Trigger = Timer] TimeLanguage (0-1)	String	Defines the language that is used to express the TimeCycle or the TimeDate value.	[+]
[Trigger = Error] ErrorCode (0-1)	String	See table 9.9 in [BPMN06] The ErrorCode attribute can be omitted if the event is located in a fault handler. In this case the error, which was caught in the fault handler, will be rethrown.	[*]

[Trigger = Compensation] Activity (0-1)	Object	See table 9.9 in [BPMN06] In BPMN+, this attribute is optional. If this attribute is not defined, all successfully completed inner scopes will be compensated.	[*]

A.8 Activity

The following table displays the common set of attributes for an activity which extends the set of common flow object attributes.

Attribute Name	Type	Description	
ActivityType	(Task | SubProcess) Task : String	See table 9.10 in [BPMN06]	[ ]
Status	(None | Ready | Active | Cancelled Aborting | Aborted | Completing | Completed) None : String	See table 9.10 in [BPMN06]	[ ]
Properties (0-n)	Property	See table 9.10 in [BPMN06]	[ ]
StartQuantity	1 : Integer	See table 9.10 in [BPMN06] This attribute is always 1 in BPMN+.	[*]
LoopType	(None | Standard | MultiInstance) None : String	See table 9.10 in [BPMN06] Since only scopes and tasks can be looping activities in BPMN+, the attribute is always None for the other activity types.	[*]
SuppressJoinFailure	False: Boolean	This attribute determines whether join failures should be suppressed when there are multiple paths leading to the activity.	[+]

A.8.1 Loop

The following table displays the attributes that can be defined for looping activities.

Attribute Name	Type	Description	
[LoopType = Standard] LoopCondition	Expression	See table 9.11 in [BPMN06]	[ ]
[LoopType = Standard] TestTime	(Before | After) After : String	See table 9.11 in [BPMN06]	[ ]
[LoopType = Multi-	(Sequential |	See table 9.12 in [BPMN06]	[ ]

Instance] MI_Ordering	Parallel) Sequential : String		
[MI_Ordering = Parallel] MI_FlowCondition	(None | One | All | Complex) All : String	See table 9.12 in [BPMN06] This attribute is always set to All in BPMN+.	[*]
[LoopType = Multi-Instance] StartCounterValue (0-1)	Expression	This is an unsigned-integer expression that the counter will be initialized with. If a participant set data object is associated with the loop, this attribute must not be specified.	[+]
[LoopType = Multi-Instance] FinalCounterValue (0-1)	Expression	This is an unsigned-integer expression that defines the final counter value. If a participant set data object is associated with the loop, this attribute must not be specified.	[+]
[LoopType = Multi-Instance] CompletionCondition (0-1)	Expression	This is an unsigned-integer expression that allows completing without executing or finishing all branches specified. If a participant set data object is associated with the loop, this attribute must not be specified.	[+]
[LoopType = Multi-Instance] SuccessfulBranchesOnly (0-1)	False : Boolean	This attribute can only be specified if the CompletionCondition attribute is set. If set to True, only branches which have completed successfully must be counted. If set to False, all branches must be counted	[+]

A.8.2 Task

The following table displays the set of attributes for a task which extends the set of common activity attributes.

Attribute Name	Type	Description	
TaskType	(Service | Receive | Send | Assign, Empty | Validate | None) None : String	See table 9.17 in [BPMN06] Only tasks of the type Service, Receive, Send, Assign, Empty, Validate and None are allowed in BPMN+. The task of type Assign, Empty or Validate are marked with a special marker.	[*]

Service Task

Attribute Name	Type	Description	
InMessage	Message	See table 9.18 in [BPMN06] In BPMN+, the message is already provided by the outgoing message flow connected to the task.	[*]

OutMessage	Message	See table 9.18 in [BPMN06] In BPMN+, the message is already provided by incoming the message flow connected to the task.	[*]
Implementation	(Web Service | Other | Unspecified) Web Service: String	See table 9.18 in [BPMN06]	[ ]
Correlations (0-n)	Correlation	The Correlations attribute is used to identify a conversation.	[+]
OpaqueInput	True: Boolean	The input variable can be opaque to hide the actual data that is sent. If this attribute is True, the ToParts must not be specified and there must not be an input variable associated with this task.	[+]
ToParts (0-n)	ToPart	The ToParts can be used instead of modeling an association from a data object which is the input variable.	[+]
OpaqueOutput	True: Boolean	The output variable can be opaque to hide the actual data that is received. If this attribute is True, the FromParts must not be specified and there must not be an output variable associated with this task.	[+]
FromParts (0-n)	FromPart	The FromParts can be used instead of modeling an association to a data object which is the output variable.	[+]

Receive Task

Attribute Name	Type	Description	
Message	Message	See table 9.19 in [BPMN06] In BPMN+, the message is already provided by the incoming message flow connected to the task.	[*]
Instantiate	False: Boolean	See table 9.19 in [BPMN06]	[ ]
Implementation	(Web Service | Other | Unspecified) Web Service: String	See table 9.19 in [BPMN06]	[ ]
Correlations (0-n)	Correlation	The Correlations attribute is used to identify a conversation.	[+]
OpaqueOutput	True: Boolean	The output variable can be opaque to hide the actual data that is received. If this attribute is True, the FromParts must not be specified and there must not be an output variable associated with this task.	[+]

FromParts (0-n)	FromPart	The FromParts can be used instead of modeling an association to a data object which is the output variable.	[+]
MessageExchange (0-1)	String	This attribute defines a message exchange name that can be used to disambiguate the relationship between the receive task and following send tasks.	[+]

Send Task

Attribute Name	Type	Description	
Message	Message	See table 9.20 in [BPMN06] In BPMN+, the message is already provided by the outgoing message flow connected to the task.	[*]
Implementation	(Web Service | Other | Unspecified) Web Service: String	See table 9.20 in [BPMN06]	[ ]
Correlations (0-n)	Correlation	The Correlations attribute is used to identify a conversation.	[+]
FaultName (0-1)	String	This attribute can be specified to indicate that a fault message is replied. The fault name can only be specified if the send task is connected with a service task by a message flow.	[+]
MessageExchange (0-1)	String	This attribute defines a message exchange name that can be used to disambiguate the relationship between receiving activities and the send task.	[+]
OpaqueInput	True: Boolean	The input variable can be opaque to hide the actual data that is replied. If this attribute is True, the ToParts must not be specified and there must not be an input variable associated with this task.	[+]
ToParts (0-n)	ToPart	The ToParts can be used instead of modeling an association from a data object which is the input variable.	[+]

Assign Task

Attribute Name	Type	Description	
Validate	False : Boolean	The validate attribute specifies to validate all variables modified by the Copy attribute.	[+]
Copy (1-n)	Copy	This attribute is used to copy data from one variable to another, as well as to con-	[+]

		struct and insert new data using expressions.	

Empty Task

The task of type Empty has no additional attributes.

Validate Task

The task of type Validate has no additional attributes.

A.8.3 Sub-Process

The following table displays the common set of attributes for a sub-process which extends the set of common activity attributes.

Attribute Name	Type	Description	
SubProcessType	(Embedded | Independent | Reference) Embedded : String	See table 9.13 in [BPMN06] This attribute always has the value Embedded in BPMN+.	[*]
EmbeddedSubProcessType	(Scope | Handler) Scope : String	BPMN+ introduces the new Sub-Process types Scope and Handler.	[+]
IsATransaction	False : Boolean	See table 9.13 in [BPMN06] This attribute is always False in BPMN+.	[*]
[SubProcessType = Embedded] GraphicalElements (0-n)	Object	See table 9.14 in [BPMN06]	[ ]
[SubProcessType = Embedded] AdHoc	False: Boolean	See table 9.14 in [BPMN06] BPMN+ does not support ad hoc sub-processes, so the value of this attribute is always False.	[*]

Scope

Attribute Name	Type	Description	
ExitOnStandard-Fault (0-1)	False : Boolean	If set to False, the process within the scope can handle standard faults using fault handler. If set to True, the process must exit immediately when a standard fault is encountered.	[+]
MessageExchanges (0-n)	String	This attribute defines message exchange names that can be used to disambiguate the relationship between inbound communicating flow objects and send tasks.	[+]
Isolated (0-1)	False : Boolean	If set to True, this attribute provides control of concurrent access to shared re-	[+]

		sources.	

Handler

Attribute Name	Type	Description	
HandlerType	(Fault | Message | Timer | Termination | Compensation) : String	The handler type of fault, termination and compensation handlers is determined by the attached intermediate event, the handler is connected with. A sequence flow with an attached error intermediate event leads to a fault handler. A sequence flow with an attached termination event leads to a termination handler. A compensation handler is not allowed to have any incoming or outgoing sequence flow. It is associated with an attached compensation intermediate event. Message and timer handlers are not allowed to have incoming or outgoing sequence flow. The type of these handlers is determined by the contained start event. A timer handler contains a timer start event and a message handler contains a message start event.	[+]
[HandlerType = Message or Timer] Parent	Object	This is the name of the parent process or sub-process, the handler is located in.	[+]
[HandlerType = Message or Timer] ExitOnStandard-Fault (0-1)	False : Boolean	If set to False, the process within the handler can handle standard faults using fault handlers. If set to True, the process must exit immediately when a standard fault is encountered.	[+]
[HandlerType = Message or Timer] Isolated (0-1)	False : Boolean	If set to True, this attribute provides control of concurrent access to shared resources.	[+]

A.9 Supporting Types

A.9.1 Assignment

Attribute Name	Type	Description	
To	Property	See table B.41 in [BPMN06]	[ ]
From	Expression	See table B.41 in [BPMN06]	[ ]

AssignTime (0-1)	(Start | End) Start : String	See table B.41 in [BPMN06]	[ ]

A.9.2 Entity

Attribute Name	Type	Description	
Name	String	See table B.42 in [BPMN06]	[ ]

A.9.3 Message

Attribute Name	Type	Description	
Name	String	See table B.44 in [BPMN06]	[ ]
Properties (0-n)	Property	See table B.44 in [BPMN06]	[ ]
From	Participant	See table B.44 in [BPMN06] In BPMN+, a message is always be related with a message flow or an activity that is connected with a message flow. This message flow can be used to determine the source of the message.	[*]
To	Participant	See table B.44 in [BPMN06] In BPMN+, a message is always related with a message flow or an activity that is connected with a message flow. This message flow can be used to determine the target of the message.	[*]

A.9.4 Object

Attribute Name	Type	Description	
Id	String	See table B.45 in [BPMN06]	[ ]

A.9.5 Participant

Attribute Name	Type	Description	
ParticipantType	(Role | Entity) Role : String	See table B.46 in [BPMN06]	[ ]
[ParticipantType = Role] Role	Role	See table B.46 in [BPMN06]	[ ]
[ParticipantType = Entity]	Entity	See table B.46 in [BPMN06]	[ ]

Entity			

A.9.6 Property

Attribute Name	Type	Description	
Name	String	See table B.47 in [BPMN06]	[ ]
Type	String	See table B.47 in [BPMN06]	[ ]
[Type = Set] Correlation (0-1)	False : Boolean	See table B.47 in [BPMN06]	[ ]

A.9.7 Role

Attribute Name	Type	Description	
Name	String	See table B.48 in [BPMN06]	[ ]

A.9.8 Import

Attribute Name	Type	Description	
Namespace	String	This attribute identifies the definitions of an imported file.	[+]
Location (0-1)	String	This attribute specifies the location of the imported file with the relevant definitions.	[+]
ImportType	String	This attribute identifies the type of document being imported. The type is specified by an URI that identifies the encoding language used in the document (e.g. "http://www.w3.org/2001/XMLSchema" or "http://schemas.xmlsoap.org/wsdl/").	[+]
Prefix	String	The Prefix is used to identify the namespace of the definitions.	[+]

A.9.9 Copy

The following table displays the set of attributes of a copy type.

Attributes		Description	
KeepSrcElementName (0-1)	False : Boolean	This attribute specifies whether the name of the destination will be replaced by the name of the source.	[+]
FromSpec	FromSpec	The FromSpec specifies the source of the copy operation.	[+]

ToSpec	ToSpec	The ToSpec specifies the target of the copy operation.	[+]

A.9.10 FromSpec

The following table displays the set of attributes of a from-spec type.

Attribute	Type	Description	
FromSpecType	(Variable| Var-Property | Ex-pression | Lite-ral | Opaque | Empty) Empty : String	This determines the type of the FromSpec. The value can be selected from a variable, a variable property, an expression or a literal value. The type Opaque means that the actual value selection is hidden. The Empty type means that the FromSpec is empty.	[+]
[FromSpecType = Variable or VarProperty] VariableName	String	The name of a defined variable that the value will be selected from.	[+]
[FromSpecType = Variable] Part (0-1)	String	This attribute denotes the part name containing the value to select. It must match an existing part name of the variable. Thus, it is only set if the variable has the type MessageType.	[+]
[FromSpecType = Variable] Query (0-1)	Query	Optionally a query can be specified to select a value from the source variable or a message part.	[+]
[FromSpecType = VarProperty] Property	String	This attribute denotes the property name containing the value to be assigned to the variable. It must match an existing property defined for the variable.	[+]
[FromSpecType = Expression] Expression	Expression	This attribute defines the expression that returns a value.	[+]
[FromSpecType = Literal] Literal	String	This attribute defines a literal value that will be assigned to the variable.	[+]

A.9.11 ToSpec

The following table displays the set of attributes of a ToSpec type.

Attributes		Description	
ToSpecType	(Variable| Var-Property | Ex-pression | Em-	This determines the type of the ToSpec. The target of the assignment can be a variable, a variable property or an expressi-	[+]

	pty) Empty : String	on. The Empty type means, the ToSpec is empty.	
[ToSpecType = Variable or VarProperty] VariableName	String	The name of a target variable.	[+]
[ToSpecType = Variable] Part (0-1)	String	This attribute denotes the target part name that the value will be assigned to. It must match an existing part name for the variable. Thus, it is only set if the variable has the type MessageType.	[+]
[ToSpecType = Variable] Query (0-1)	Query	Optionally a query can be specified to select a value from the target variable or a message part.	[+]
[ToSpecType = VarProperty] Property	String	This attribute denotes the property name the value will be assigned to. It must match an existing property defined for the variable.	[+]
[ToSpecType = Expression] Expression	Expression	This attribute defines the expression to select a value.	[+]

A.9.12 Query

The following table displays the set of attributes of a Query type.

Attributes		Description	
Language (0-1)	String	This attribute specifies the language of the query.	[+]
Content (0-1)	String	The Content attribute defines the expression to select a value. If no content is specified, the query is supposed to be opaque.	[+]

A.9.13 Correlation

The following table displays the set of attributes of a Correlation type.

Attributes		Description	
Set	String	This attribute must match a correlation set name that is specified in the enclosing process.	[+]
Initiate	(Yes | No | Join) No : String	The Initiate attribute determines if the related activity must attempt to initiate the corresponding correlation set. If set to Join, it must only attempt to initiate the correlation set if it is not yet initiated.	[+]

Pattern (0-1)	(Request | Re-sponse | Re-quest-Respon-se) : String	The Pattern attribute must be specified if the correlation belongs to a task of type service. It is used to indicate whether the correlation applies to the request message, the response message or both.	[+]

A.9.14 FromPart

The following table displays the set of attributes of a FromPart type.

Attributes		Description	
Part	String	The Part attribute references the part of the message that the value will be taken from. The value "##opaque" means to hide the actual part.	[+]
ToVariable	String	The ToVariable indicates the variable name that the value will be copied to. The value "##opaque" means to hide the actual variable.	[+]

A.9.15 ToPart

The following table displays the set of attributes of a ToPart type.

Attributes		Description	
Part	String	The Part attribute references the part of the anonymous temporary WSDL variable the variable value will be copied to. The value "##opaque" means to hide the actual part.	[+]
FromVariable	String	The FromVariable indicates the variable name the value will be copied from. The value "##opaque" means to hide the actual variable.	[+]

A.9.16 Expression

The following table displays the set of attributes of an Expression type.

Attributes		Description	
ExpressionLanguage (0-1)	String	This specifies the language for the expres-sion.	[+]
Expression (0-1)	String	See table B.43 in [BPMN06] In BPMN+, this attribute is optional. If this attribute is not set, the expression interpre-ted as opaque.	[*]

A.9.17 Gate

The following table displays the set of attributes of a Gate type.

Attributes		Description	
OutgoingSequence-Flow	SequenceFlow	This is the sequence flow associated with the gate. Its Condition attribute depends on the gateway, the gate belongs to.	[+]

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Kerstin Pfitzner)