Checking the Enforceability of BPMN Choreography diagrams and generating Colaboration diagrams from them

tmi (thomas.milde@student.hpi.uni-potsdam.de)

This "documentation" will state the basic ideas of how I implemented enforceability checking and generating behavioral interfaces. For details, please refer to the code.

1 Enforceability checking

There are several conditions stated by the BPMN-specification, that have to be fulfilled in order for a choreography to be enforceable and some conditions had to be added aditionally.

There are several conditions, that can be checked in a very simple way:

- Only allowed nodes are present: not all BPMN-nodes may occur in BPMN-diagrams
- Attached events may only be attached to ChoreographyTasks, not to ChoreographySubProcesses
- EventBasedGateways may only be followed by TimerIntermediate- and SignalIntermediateEvents and ChoreographyTasks
- Only one initiating and one other Message may be attached to each ChoreographyTask. This condition is not stated in the specification, but there is no definition, what semantics more than one associated message of each type could have and therefore the decision was taken to forbid this case.
- The initial participant of any ChoreographyActivity is not a multi-instance participant. This either was not stated in the specification, but I think, that the semantics of this would be unclear. **This is questionable. I or someone else will have to think about it again.**

In the next sections, some more complicated checks will be described.

1.1 The initiator of any task must be a participant of every preceding activity

If this condition would not hold, the initiator of a task could not know, when to send the initiating message, because he cannot know, wheter all prior activities have been finished.

In order to find out all preceding activities, it is necessary to search in the opposite direction of all incoming SequenceFlow until an Activity was found. Because there may be loops in the model, it is not possible to do this simply by recursively scanning the incoming SequenceFlow-edges. Therefore the structure `BranchingTree` (see 1.5) is used. It enables finding all activities, from which leads a path, that contains no activity, to the activity currently being analyzed. The example of figure 1 shows, that this approach is not sufficient, because it would lead to task0 being considered as a preceding node of task2, allthough it cannot be a direct predecessor. In order to detect this case, for every node, that would be treated as a predecessor of task2 (i.e. task0 and task1), it would be checked, wheter its outgoing path splits into multiple parallel pathes, that merge again before reaching task2, after at least one path contained an activity. If this condition holds, this node (in the example: task0) is not a predecessor.

1.2 Checking for situations, where participants wait indefinitely at EventBasedGateways

The BPMN-specification states "A time-out may be used to ensure that the Gateway does not wait indefinitely" (BPMN-specification 2.0 ... page 344), but actually a timeout does not realize the semantics of the choreography. Therefore the EnforceabilityCheck will produce a warning, when this situation occurs.

In order to check for such situations, every EventBasedGateway is analyzed: If a participant is not the initiator of at least one of the directly following tasks, he will know the decision of the gateway only from the messages he receives. That means, that he either has to receive a message in every possible path from the gateway to the end of the choreography or he has to receive no more messages after the gateway. This condition does not need to hold, if the participant was not participating in the choreography before the gateway was reached, because in this case, the messages he receives are initial messages for him then.

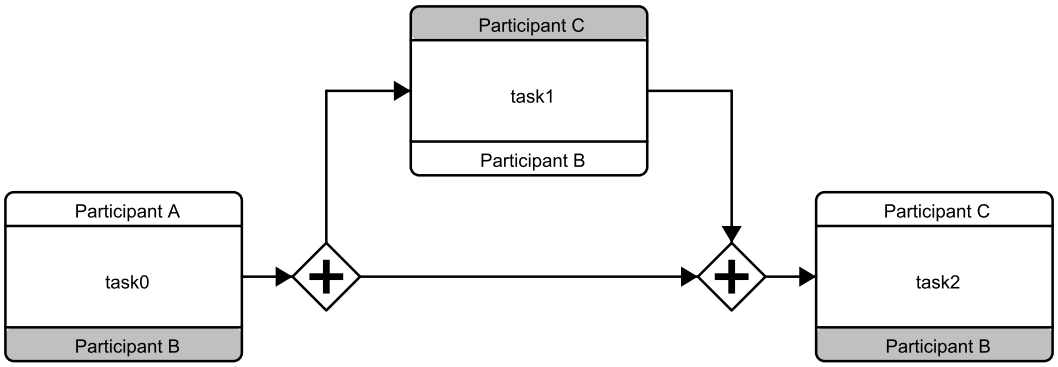

Figure 1: Determining the preceding tasks of task2 needs some more checking than just following the incoming SequenceFlow

1.3 Every participant, who is affected by the decision of an Inclusive gateway, must share the data, on wich the decision is based

Inclusive gateways can only be realized as such, even for participants, who are only receiving messages, because there is no way of realizing inclusive behavior with EventBasedGateways.

In the model-representation used by the workbench it is practically not possible to strictly check, wheter all participants know the data, on wich the decision is based, because there is no way of formally defining data and conditions for gateways. Therefore, it is only checked, that every participant, who is affected by the gateway has been participating in the colaboration before it reached the gateway.

A participant is not affected by the gateway, iff the next following activities, that involve him, are the same for every outgoing SequenceFlow of the gateway. This is checked by building a `BranchingTree` for the outgoing flow of each directly succeeding node of the gateway and identifying the first nodes, that involve a participant (for each participant).

1.4 Architecture of the enforceability check

Each of the checking-aspects mentioned before is realized as a concrete subclass of AbstractChoreographyCheck, which provides some helper methods, that are usefull for multiple Checks and declares the abstract methods `checkObject (ProcessObject)` and `getRelevantClasses()`. The method `getRelevantClasses` returns a collection of class-objects (that must be subclasses of `ProcessObject`). These are the classes of objects, that the concrete subclass is interested in, i.e., that it wants to check. This checking of one object is done by the method checkObject, which returns a collection of `EnforceabilityProblems`. An `EnforceabilityProblem` is a structure, that holds a text, a `ProblemType` (`enum`, that describes the type of problem - an error or a warning because of a necessary timeout), a primary object (the `ProcessObject`, that is mainly responsible or most significant for the problem) and a set of other related `ProcessObjects`. If `checkObject` does not find any problems, it will return an empty collection.

The subclasses of `AbstractChoreographyCheck` are combined to a complete enforceability check by `CombinedEnforceabilityChecker`. In order to do this, it iterates over all `ProcessObjects` in the choreography and collects the problems from all `AbstractChoreographyCheck`-subclasses, that contained to object´s class (or one of its superclasses) in the result of their `getRelevantClasses`-method.

The class `EnforceabilityPlugin` is a `WorkbenchPlugin`, that provides the enforceability check to the user. It graphically highlights the problems as stated in the `EnforceabilityProblem`-objects by adding a `TextAnnotation` to the main object of each problem and highlighting all related objects and the main object.

1.5 The data-structure `BranchingTree`

A `BranchingTree` is a loopless representation of the structure of an excerpt of a BPMN-diagram. It is mainly intended to be used for choreographies, but it is also usable for colaborations, what is done in a few places. A `BranchingTree` may be created to represent the sequence flow or the inverted sequence flow. It consists of four types of inner nodes: `ActivityNodes` represent a `ChoreographyActivity` which is not an implicit split or join; `SilentNodes` represent all other nodes, that are no split- or join-nodes; `ParallelSplit` represents a `ParallelGateway` (even if this is not a split - in this case it will have just one child node) or any node, that is an implicit split (a node, that is not a gateway, but has

multiple outgoing sequence flow of wich none is conditional, but only if the tree has the same direction as the sequence flow, there are no implicit splits in the opposite direction); `AlternativeGateway` represents any other type of gateway, or any node, that is an implicit split (i.e. any node, that is not a gateway, and has multiple incoming SequenceFlow, if the tree is built in the opposite direction of the sequence flow, or a node, that has multiple outgoing sequence flow of wich at least one is conditional flow, if the tree is built in the direction of the sequence flow). Additionally, there are the nodes `BranchingTreeRoot`, which is necessary, because every other node can rely on having a parent node, wich it knows, and `EmptyBranchingTree`, which forms the other end of the tree.

To build a `BranchingTree`, the class `TreeBuilder` must be used. It enables, like already mentioned, to build a tree following the sequence flow in its own direction or reversed and starts building from one node, that is to be specified in the call to `buildTreeFor`. The resulting `BranchingTree` can perform operations like checking, wheter all of its alternative pathes contain `ChoreographyActivities`, that involve a specific participant or checking, wheter all parallel pathes, that originate from this tree merge again before a specific node is reached. Due to the fact, that it is used for numerous specific tasks, there is a great range of methods offered.

1.5.1 Implementation of the `TreeBuilder`

In order to build a `BranchingTree`, the `TreeBuilder` creates a set of "active" `BranchingTree`-nodes, i.e. these nodes, to wich still children have to be added. At the beginning, this contains only the node for the `ProcessNode`, at wich the tree was requested to start. As long as there are active nodes left, the TreeBuilder continues its work, which consists of iterating over the activeNodes and generating new `BranchingTree`-nodes for each following (or preceding – according to the direction, in wich the tree is being built) `ProcessNode` of the node, wich the active `BranchingTree`-node represents. If a `ProcessNode` is met, wich already occured more than one time on the path from the root to the active ndoe, the alternative path of the active node will be removed, what means cutting it off at the last `AlternativeSplit`-node. This ensures, that the algorithm comes to an end, i.e. does not endlessly iterate in loops of the `BPMNModel`, but loops are not completely omitted – one iteration of each loop will be present.

2 Generating Colaboration diagrams

A pattern-based approach was chosen to generate a colaboration diagram from a choreography. This means, that, in order to generate colaborations, every element of the choreography is translated into elements of the colaboration. It seems not to be possible to do this with just a local view on the object currently being translated. Some operations, for example, have to find out, wheter a node could be the first interaction of a participant or not. After the pattern-based translation was done there needs to be done some correcting and improving work on the diagram. The following sections will explain the single steps of generating a colaboration.

2.1 Generating Pools

First of all, a pool for each participant has to be created. This can be done by simply iterating over all nodes of the choreography and collecting the participant. It must also be looked up, wheter a participant occurs as a multi-instance participant in some Choreography Activity. In this case, the pool has to be marked with an MI-marker. At this point, no respect is paid to the user´s selection of wich pools should be modelled as black-box pools. All pools will be generated in full detail and at the end, the details of pools, that were not selected by the user to be shown in detail, they will be collapsed, see section 2.9. In case of future development, this is surely a point that should be improved: the content of pools, that were not selected needs not to be generated at all.

2.2 Translating Choreography Tasks

The Choreography Tasks are the elements, that contain the actual interaction. In the pool of the initiator of the Choreography Task, a Task will be generated, that can either be a service task, if there is an answer (i.e. a not-initiating message attached to the Choreography Task), or, otherwise, a send task. The loop-type of this task will be copied from the original Choreography Task or, if the receiver of the Choreography Task is a multiple participant and the Choreography Task is no loop, will be set to Multiple Instance (parallel).

The situation is more complex in the pool of the receiver. Here it must be distinguished between a situation, where the Choreography Task is the first participation of the receiver, situation, where he has participated in the choreography at a prior point, and situations, where both is possible. Prior participation means that he was a participant of a Choreography Task, that was executed prior to the one that is being translated now, or that there was a non-empty Start Event (i.e. a Timer, Conditional, Signal or Multiple Start Event) prior to this Choreography Task. The second is considered as participation, because the occurence of the Start Event´s trigger told him, that the choreography will be executed. In contrast, if he did not participate so far, he knows only from the first message, wich he receives, that the choreography is being executed. Additionally it has to be differentiated between Choreography Tasks, that are loops and those, that are not.

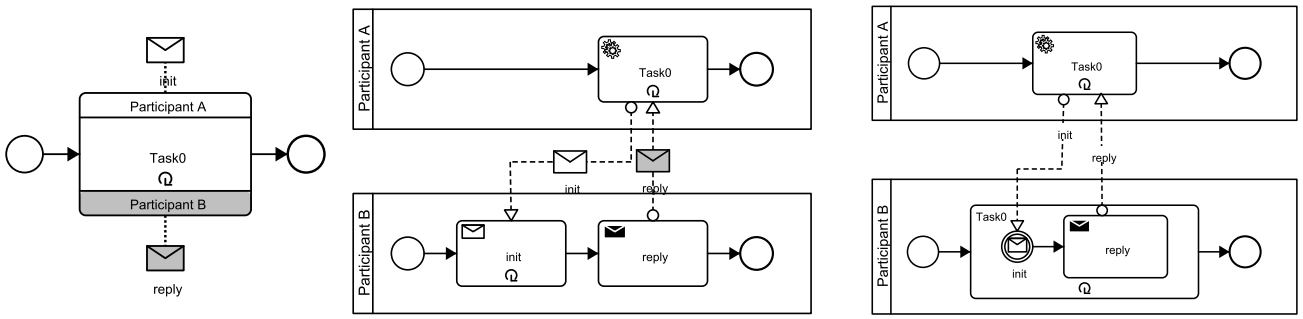

Figure 2: middle: what the current implementation generates from the choreography (left) and right: what should be generated

If the Choreography Task may be the first participation of the receiver, a Start Event has to be set up. More exactly, a Message Start Event is used, if the Choreography Task is no loop, otherwise an empty Start Event will be used, because the loop cannot be represented in an Event and therefore a task must be used for receiving the message. If the receiver participated prior to the Choreography Task, an intermediate event is generated for receiving the message, if there is no loop. If there is a loop, receiving is done by a receive Task with the loop marker copied from the choreography (no matter, wheter there there was prior participation). If the Choreography may be the receiver´s first participation, but is not necesarily, the flow from the start event and the incoming flow has to joined. If the Choreography Task is a loop, this is done by an Exclusive gateway, wich is set up in front of the Receive Task and joins the incoming flow with the flow from the generated start event. If it is not a loop, an Exclusive Gateway will be set up, that joins the flow of the Start Event and the Intermediate Event. If there is an answering Message associated with the Choreography Task, a send-Task will be generated in the receiver´s pool following the last of the nodes generated before (e.g. the looped receive-Task or the Exclusive Gateway, if there is no loop). Note that there is a **BUG** here (see figure 2): If a looped Choreography Activity has a reply, only the receive Task is realized as loop, but actually a Sub Process should be set up, that contains the receive- and send-nodes and has a loop marker. Due to this being a large change in the class. The last step of translating the Choreography Task itself is to add Message Flow between the corresponding nodes. `TaskTranslator`, there is not enough time left to implement this.

2.2.1 Translating attached Events

Translating a Choreography Task also involves translating the attached Events at this node. There are two cases: an attached Event can be a Message Intermediate Event or another Intermediate Event. The first thing to be looked at are the other Intermediate Events (Timer, Cancel, Compensation, Conditional, Signal or Multiple Intermediate Events). Their occurence cancels the execution of the Task for both participants and redirects the choreographie´s execution to another Sequence Flow than the one, that would usually be chosen after the Choreography Activity. In the initiator´s pool the attached Event is translated one-to-one: A copy of the Choreography Activitie´s attached Event is attached to the receiver´s task. Again, the situation is slightly more complex in the receiver´s pool: If the Choreography Task may be the receiver´s first participation in the choreography, a Start Event with the same trigger type as the attached Event will be generated. If it is not necessarily the first participation of the receiver, an Intermediate Event will be generated (a copy of the attached Event). If both were generated, their outgoing Sequence Flow will be joined with an Exclusive Gateway. If the Intermediate Event was generated, additionally an Event Based Gateway must be added, that has outoing Sequence Flow to the new Intermediate Event and the receive-Event or -Task for the initial message. If there is a reply, a copy of the Event also has to be attached to the receiver´s send Task. A new Exclusive Gateway will be added, that joins the outgoing Sequence Flow of this newly attached Event and the Event(s) that were added before. This can lead to two Exclusive Gateways being set up one directly succeeding the other, but this will be improved at the end (see section 2.8). It would be more complicated to avoid setting up this situatiion, than letting this happen and improving it later, particularly because there are more points in the process of generating the colaboration, where such situations may be created.

2.2.2 Data-structures for identifying the generated nodes

When a node is generated, it is always added to the map `allNodes`, which maps from the ID of a node in the Choreography diagram to a mapping from the name of a participant to a collection, that contains all the nodes, wich were added to the pool of the participant in order to realize the node with the given ID. This is necessary in order to generate Sub Processes, see section 2.5. Additionally, there are the fields `firstNode`, which maps from the choreography-node-ID and the participant name to the first node, that realizes the ID, and `lastNode` (analogous). The first node is the node, to wich Sequence Flow, that lead to the choreography node, should lead in the colaboration;

the last node is the one, from where outgoing Sequence Flow should lead away. These two fields are required to realize the Sequence Flow between different nodes of the choreography (see section 2.6).

2.3 Translating Events

In the next step all Events, except attached Events (wich have already been translated), will be realized in the colaboration. Empty Start Events will be ignored, because they are just optical markers, that show the beginning of the choreography, but the participants´ processes are instantiated when they receive the first message, except the processes of participants, who send the first messages (for those, Start Events will be added while generating the Instantiating Gateways, see section 2.7). Non-Empty Start Events and End Events will be copied to the pools of all participants. Intermediate Events require more fine-grained considerations, because there are situations, where Intermediate Events in the choreography must be realized as Start Events. For each participant it must be found out, wheter the Intermediate Event could be a Start Event for the participant (this is the case if the participant has not participated in the choreography before) and wheter it could be intermediate for him (he may have participated before) – again, both can be true. If it may be a Start Event, such a node, with the same trigger as the Intermediate Event, will be generated in the participant´s pool. If it may be intermediate, the Intermediate Event will be copied to the participant´s pool and if both is possible, an Exclusive Gateway will be generated, that joins the outgoing Sequence Flow of the Start Event and the Intermediate Event.

2.4 Translating Gateways

Parallel, Inclusive and Complex Gateways are copied to the pools of all participants. For Exclusive and Event Based Gateways two cases have to be distinguished: For the initiators of Choreography Activities, that directly follow the Gateway, an Exclusive Gateway has to be used and in the pools of all other participants an Event Based Gateway is set up. This means, there is exactly no difference between the realization of an Exclusive Gateway and the realization of an Event Based Gateway. See section 3.1 for some more thought about this aspect.

2.5 Creating Sub-Processes

The contained nodes of Choreography Sub-Processes were translated in the prior sections since they are also contained in the choreography model, but the Sub-Process itself has not yet been translated into the colaboration. A Sub-Process is generated in the pool of every participant of the Choreography Sub-Process and all nodes, that are concerned with realizing one of the Choreography Sub-Process´ contained nodes (see section 2.2.2, field `allNodes`) are moved to the Sub-Process. In order to have generated all contained nodes of a Sub-Process before it is generated itself, translating a Choreography Sub-Process will be postponed until all of its contained Choreography Sub-Processes have been translated.

2.6 Generating Sequence Flow

In order to realize the Sequence Flow relationship defined by the choreography every Sequence Flow edge is taken into consideration for every participant. If there is no entry for the source of the edge and the current participant in `lastNode` (see section 2.2.2), what means, that the source was not realized in the current participant´s pool due to him not being a participant of this node,the edge will be ignored for this participant. If there is an entry for the source and also one for the target a new Sequence Flow edge will be added to the colaboration diagram between these two nodes, that were found in `lastNode` and `firstNode`. If there is an entry for the source, but not for the target, a `BranchingTree` (section 1.5) will be built for the target following its outgoing Sequence Flow. From this tree, the first node on each path, that has an entry in `firstNode` for the current participant, will be collected and a new Sequence Flow from the source of the original Sequence Flow to each of the collected nodes will be added.

2.7 Generating Instantiating Gateways and remaining Start Events

Instantiating Gateways are not strictly necessary, but they help in understanding the model. First all nodes (except catching Intermediate Link Events) of the choreography without incoming Sequence Flow are located. They are the points, where the choreography can start (it is not strictly necessary to place Start Events). For each of this nodes a `BranchingTree` is built. The `BranchingTree` provides the functionality of creating instantiating gateways and missing Start Events. This is done by recursively stepping down the tree and creating an instantiating gateway at every split (a parallel instantiating gateway for a `ParallelSplit`-node and an exclusive instantiating gateway for an `AlternativeSplit`-node). The recursion stops, when a node was found, in wich the requested participant is involved or when a Non-Empty Intermediate or Start Event was found. At this point, a Start Event is created in the participant´s pool, if it is not already created (can be determined with the map `startEvents`, see section 2.2.2). The recursion also stops, when a parallel merge is reached that joins with a path, on wich the participant already was involved (this can happen, if there is a structure like in figure 1).

2.8 Correcting the diagram and improving its readability

So far all nodes and edges have been translated, but there are errors and stylistic flaws (constructions, that are allowed, but are not well readable or are discouraged) in the diagram. Therefore, correcting work needs to be done. First of all, incoming SequenceFlow of Start Events is removed, unless its source is an instantiating Gateway. The next three steps will be repeated until no more changes have been done to the model, because each step can require another step to do further work. The steps are:

Removing needless edges If multiple Sequence Flow edges lead from one gateway to another gateway of the same type or to an Inclusive Gateway or from an Exclusive Gateway to an Event-based Gateway or the other way around, they can be reduced to one edge without changing the semantics of this diagram. If a Sequence Flow edge leads from a Parallel Gateway to another Parallel Gateway and there is another path between these two gateways (wich may include other nodes), this edge can be removed.

Removing needless gateways If a Gateway has at most one incoming Sequence Flow edge and at most one outgoing Sequence Flow edge, it can be removed: Its incoming Sequence Flow´s target is set to the outgoing Sequence Flow´s target (if it has incoming and outgoing Sequence Flow; otherwise the incoming edge is removed, if there was one), the outgoing edge is removed (if there was one) and the Gateway is removed from the model.

Uniting consecutive gateways Two consecutive gateways can be united, if they have the same type (or one is Exclusive and the other is Event-based (and not parallel instantiating)), but only if the first gateway is a pure join (has only one outgoing Sequence Flow) or the second is a pure split (has only one incoming Sequence Flow).

2.8.1 "Pulling" events to Event-based Gateways

Event-based Gateways may only be followed by Intermediate Events or receive-Tasks, but due to the way, how the colaboration was generated, it happens, that an Event-based Gateway is followed by other Gateways, wich are followed by Intermediate Events or other Gateways again. Finally the chain of Gateways will lead to an Intermediate or End Event or a dead end. In case it leads to the end of the process, a timeout will be set up later. First only the case, that it leads to an Intermediate Event, will be considered. The gateways, wich can be dealt with, are Exclusive and Event-based gateways, Parallel fork (only one incomming edge!) Gateways and theoretically (this is not implemented - It seems, that I forgot this case) Inclusive split (also only one incomming edge) Gateways. Figures 2.8.1, 2.8.1 and 2.8.1 show, how pulling events to the Event-Based Gateway is done.

After the Event-Based Gateways have been corrected as far as possible, again the three steps mentioned before are done in a loop until no more changes occur, but there are two additional operations, that are also enacted in this loop:

Removing nodes without incoming Sequence Flow Nodes, except Start Events and instantiating gateways, without incoming Sequence Flow can be eliminated, because Start Events and Instantiating gateway are the only nodes, where the process should start since they were generated for all initial nodes (see section 2.7), but when "pulling" events to Event-Based gateways some nodes may loose their incomming edges and therefore become obsolete.

Eliminating Duplicates Through the prior "event-pulling" it happens at some points, that there are two edges leading from an Event-Based or Exclusive Gateway to a timeout or message-receive (both lead to seperate timeout-Events or both lead to seperate catching Message Intermediate Events for the same message) and from both timeouts/message-receives an edge leads to an Event-Based or Exclusive gateway (actually, there may be even more than two of these duplicates). Only one of this pathes needs to be kept in the diagram, the other(s) can be removed.

At the end four more steps are performed (the first entry in the following list consists of two steps), but just one time each:

Removing implicit splits and joins Implicit joins are nodes, that are not gateways, with multiple incoming Sequence Flow. This is allowed and has the meaning of an exclusive join, but modellers may not like this style (like I do) and therefore implicit splits and joins are removed, if the user selected this option. Removing implicit joins also contains removing joins at Event-Based Gateways, because there is no special semantics defined for joining at Event-Based Gateways.

Removing obsolete fragments Fragments (sets of nodes, that do not have any Sequence Flow connection to other nodes) are considered being obsolete, if they contain no Task, nonempty Cluster, Message Intermediate Event, Message Start Event, or throwing Intermediate Event, because a Process consisting only of other nodes cannot contain any interaction.

Generating timeouts There may be the necessity of generating timeouts to avoid waiting indefinitely at Event-Based Gateways (see section 1.2). They are generated on every node, that leads from an Event-Based Gateway to a node, that is not an Intermediate Event or (receive-)Task.

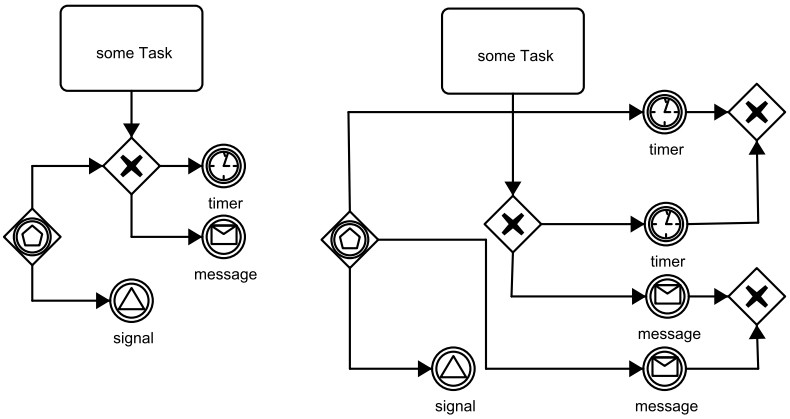

Figure 3: Left: an Exclusive Gateway is directly following an Event-based Gateway; Right: how this will be corrected

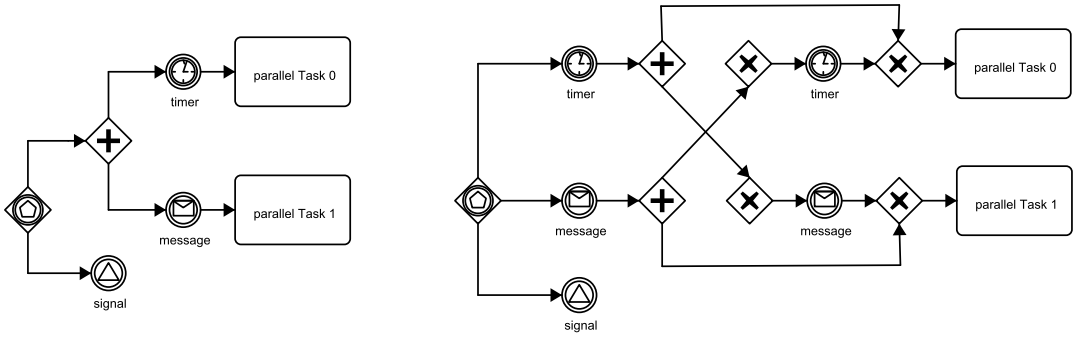

Figure 4: Left: a Parallel Gateway is directly following an Event-based Gateway; Right: how this will be corrected, the Exclusive Gateways in front of message and timer were inserted, because there could be more than two parallel pathes and in this case there would be more than one incoming Sequence Flow to them

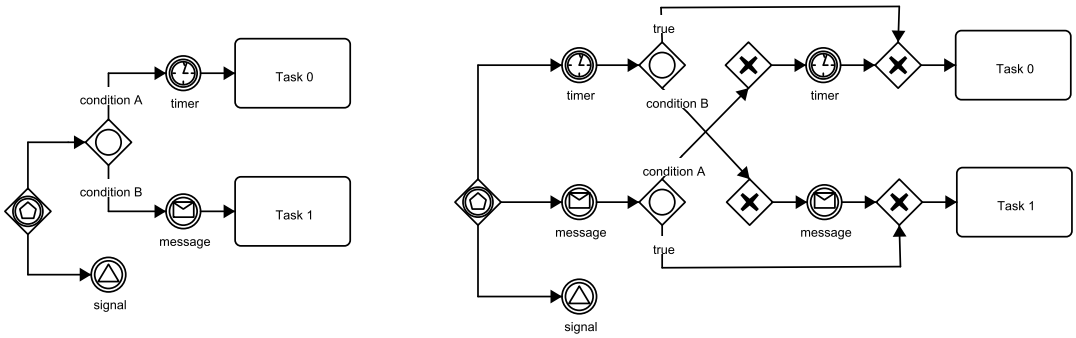

Figure 5: Left: an Inclusive Gateway is directly following an Event-based Gateway; Right: how this could be corrected (lacks implementation), the Exclusive Gateways in front of message and timer were inserted, because there could be more than two outgoing edges at the Inclusive Gateway and in this case there would be more than one incoming Sequence Flow to them

2.9 Hiding the details of pools, that have not been selected to be modelled in full detail

As last action (let away layouting, what is not done by me) those pools will be collapsed, wich are not selected to be modelled in full detail. First all incomming and outgoing Message Flow to/from nodes in the Pool is redirected to the Pool itself and the activities are removed from the Pool and diagram. At last the collapsed-attribute of the Pool is set to true.

3 Problems and Vaguenesses on Choreography diagrams and the automatic generation of Behavioral interfaces

3.1 How to realize exclusive Gateways

The specification is inconsistent with itself in this point: In the examples, that it gives, it shows, that exclusive gateways are realized as exclusive gateways in the pools of all participants, even in the pools of those participants, who only receive messages. On the other hand, it states "For the receivers in Choreography Activities that follow the Gateway, an Event-Based Gateway is to consume the associated Message" (page 342). As of now, I decided to use this (the second) alternative, but this leads to the situation, that there is no difference between an event based Gateway and an exclusive Gateway.

3.2 Inclusive Gateways in the Choreography

An Inclusive Gateway can only be realized as an Inclusive Gateway in the pool of every affected participant, because there is no way of realizing inclusive behavior with event based decisions. That means, that every participant, who is affected by the gateway´s decision, has to know the data, on wich the decision is based. In workbench-models there is no way of formally defining data or gateway conditions and if there would be the possibility of doing so, it is very probable, that it would not be used in most cases. So the only thing, that can be checked is, wheter all participants, who are affected by the gateway´s decision have been participating in the choreography before the Gateway was reached.

3.3 Dupplication of Message Receiving

There are several cases, when it is necessary to receive one message alternatively at different nodes:

- if a Task may be the first Acitivity for its receiver, but is not necessarily the first Activity, there must be a Message Start Event and a Message Intermediate Event, wich alternatively receive the same Message
- If there is a Parallel or Inclusive Gateway in one alternative of an Eventbased or Exclusive Gateway, before the first participation of a particular participant after the Eventbased/Exclusive Gateway and this participant receives messages in multiple pathes of the parallel gateway (see figure 6), there is no way of knowing in wich order the messages from the parallel pathes arrive and therefore there must be a duplication of the receive events for this Activity in order to offer alternatives for an event-based gateway. See figure 7 for how 6 can be realized.

3.4 Timeouts

The specification allows the usage of timeouts to avoid indefinite waiting at Event Based Gateways, but this does not reflect the semantics intended by the choreography: a choreography usually does not state any constraints according to how long it might take until a message is sent.

3.5 May Event Based Gateways be followed by a Choreography Subprocess?

It is specified, that there may be a Choreography Activity connected to an Event Based Gateway, but I think, only Choreography Tasks, no Sub-Processes should be allowed in this position, because a SubProcess does not always start with only one Choreography Task and so the semantics of this are not clear.

3.6 Multi-instance initiators

It is not forbidden for the initiator of a task to be an MI-participant. This could lead to semantical problems, but I am not completely sure about this, so this is a point, about wich should be thought.

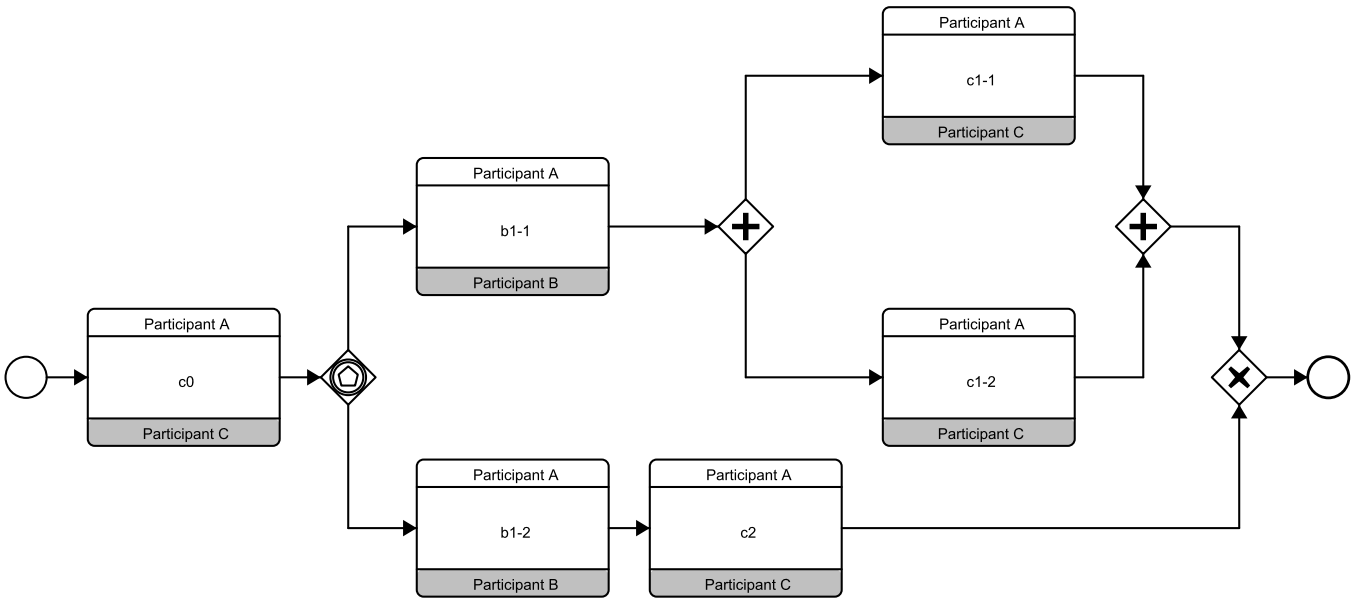

Figure 6: Receiving c1-1 and c1-2 must be duplicated in the corresponding colaboration diagram

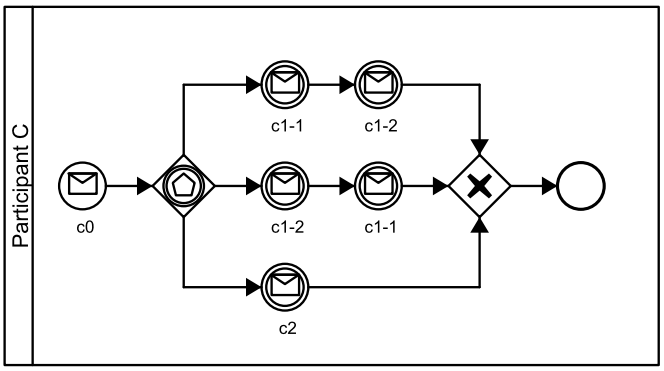

Figure 7: Behavioral interface for Participant C of the choreography shown in figure 6

3.7 Attached Intermediate Events

It is questionable, wheter the occurence of an Attached Event´s trigger could instantiate a participant´s process. In the current implementation, this is possible, but actually the participant would have to know that the Activity was reached, wich the Event is attached to, so it could be more sensefull to forbid this.

An additional problem arises when considering, that communication is asynchronous: When the last message of a Choreography Activity has been sent, but not yet been received and the trigger of an Event, that is attached to the ChoreographyActivity occurs, the receiver will react to this event, but the sender has already proceeded in his Process, so that he does not pay respect to the event.

3.7.1 Message Intermediate Events

If a Message Intermediate Event is attached to the participant band of the receiver of a Choreography Task, the message specified by this event is sent by the initiator. If he would be allowed to send it after sending the initial message of the Choreography Task, before he receives the reply, the receiver could already have sent the reply, so that they would choose different ways in the choreography (and the two messages would never be received). This is why we decided to allow sending this message only alternatively to the initial message of the Choreography Task, but we cannot imagine any practical use-case for this.

A Message Intermediate Event, that is attached to the initiator´s participant band is considered to be an alternative to the usual receiver´s answer. Here possible use-case is the exception-answer of a webservice.