BPMN Core Modeling Concepts: Inheritance-Based Execution Semantics

Egon Börger and Ove Sörensen

Abstract We define an abstract model for the dynamic semantics of the core process modeling concepts in the OMG standard for BPMN 2.0. The UML class diagrams associated therein with each flow element are extended with a rigorous behavior definition, which reflects the inheritance hierarchy structure by refinement steps. The correctness of the resulting precise algorithmic model for an execution semantics for BPMN can be checked by comparing the model directly with the verbal explanations in [OmgBpmn(2009)]. Thus, the model can be used to test reference implementations and to verify properties of interest for (classes of) BPMN diagrams.[1]

1 Introduction

The Business Process Modeling Notation (BPMN) is standardized by the Object Management Group (OMG). We explain here its main modeling concepts with a focus on the behavioral meaning of processes, based upon the currently (March 2010) available OMG document [OmgBpmn(2009)], abbreviated op.cit. As a distinctive feature we adapt a stepwise refinement technique to follow the successive detailing of the BPMN execution semantics along the inheritance hierarchy in op.cit.

We associate with each UML class diagram defined in op.cit. for the syntax of behavioral BPMN elements a description of their behavior. These descriptions make

Egon Börger
Visiting ETH Zürich, hosted by the Chair for Information Security, on sabbatical leave from Computer Science Department, University of Pisa, Italy, e-mail: boerger@di.unipi.it

Ove Sörensen
Institut für Informatik, Christian-Albrechts-Universität zu Kiel, Olshausenstraße 40, D-24098 Kiel, Germany, e-mail: ove@is.informatik.uni-kiel.de

[1] The work of the first author has been partially supported by a Research Award from the Alexander von Humboldt Foundation (*Humboldt Forschungspreis*) and partially supported by the Italian Government under the project PRIN 2007 D-ASAP (2007XKEHFA).

the natural language formulations in the standard document precise at the minimal level of rigour needed to faithfully capture a common understanding of business processes by business analysts and operators, information technology specialists and users (suppliers and customers). Such a common understanding, which must not be obfuscated by mere formalization features, is crucial to faithfully link the three different views of business processes by designers, implementors and users.

To obtain such a *precise, inheritance hierarchy based high-level description of the execution semantics of BPMN* we use the semantical framework developed in [Börger and Thalheim(2008)] for business process modeling notations and applied there to BPMN 1.0 [OmgBpmn(2006)]. Since it is based only upon standard document terms, it allows one to check by direct inspection the faithfulness of the description with respect to the verbal explanations in op.cit. On the other hand, the rigorous operational character of the description offers the possibility to use it as the reference model for testing and for comparing different implementations as well as for a mathematical analysis of properties of interest for classes of BPMN process diagrams (see [Wei(2010)]). Since the standardization process is still ongoing our BPMN model leaves all those issues open which are not (yet?) sufficiently clarified in op.cit. However our work shows that it would have been possible to provide a succinct and complete, rigorous and thereby objectively checkable BPMN execution semantics, although the OMG standardization committee seems to have voted against such an endeavor (op.cit.Ch.14) in favor of an informal description with various loose ends implementations will have to clarify.

Technically speaking we assume the reader to have an understanding of what it means to execute simultaneously finitely many transition rules of form

if *Condition* **then** *Actions*

prescribing a set of actions to be undertaken if some events happen; happening of events is expressed by conditions becoming true. For a simple foundation of the semantics of such rule systems, which constitute Abstract State Machines (ASMs) and can be viewed as a rigorous form of pseudo-code, we refer the interested reader to [Börger and Stärk(2003)]. Such rules are inserted as behavioral elements at appropriate places in the BPMN class hierarchy. The ASM refinement concept supports strictly following the inheritance steps in the BPMN class hierarchy. In Sect.2 we describe the class hierarchy of BPMN elements, focussing on message flow and the behaviorally relevant diagram structure, which is represented by the so-called sequence flow of flow nodes. In Sect.3- 5 we describe the three main subclasses of the BPMN *FlowNode* class, namely for gateways, activities and events. To avoid repetitions, we frequently rely upon the explanations or illustrating diagrams in the standard document and assume the reader to have a copy of it at hand.

2 Structure of the Class Hierarchy of BPMN 2.0

We restrict our attention to those features of the BPMN class hierarchy which are relevant for the behavioral description of single processes, namely diagram structure, flow elements and message flow. The class *FlowElement* in op.cit.Fig.8.23 contains, besides *SequenceFlow*s and *FlowNode*s also *Dataobject*s, which we represent by ASM locations. Their read/write operations represent what is called a "data association execution"(op.cit.Fig.10.63). Due to space limitations we investigate the single process view (called orchestration) and treat process interaction features—the collaboration of and the communication between processes, called choreography—in terms of abstract interface conditions.

2.1 Message Flow

The interaction between multiple processes happens in BPMN via communication (messages between pools, activities and events) or shared data. The concept of monitored locations in the ASM framework provides an exact interface of process instances to message handling which abstracts in particular from the BPMN choreography diagrams op.cit.Sect.12 and in particular from correlation issues for the delivery of messages. Consider an abstract operation $\textsc{Send}(\mathit{payload}(m), \mathit{receiver}(m))$ which is refined for all related elements of the BPMN *MessageFlow* class diagram op.cit.Fig.8.38; we write *sender* for *sourceRef* and *receiver* for *targeRef*. The operation is restricted by the stipulation that the receiver of a message is either a participant (here appearing as pool) or an activity or an event. Thus message arrival is reflected as an update by $\mathit{payload}(m)$ of a location which is monitored by $\mathit{receiver}(m)$; reading a message means to read such a monitored location.

2.2 Diagram Structure (Sequence Flow)

The BPMN diagram structure is used to pictorially represent a business process and is defined by the *SequenceFlow* class diagram op.cit.Fig.8.48. The sequence ('control') flow shows the order of flow elements in a process. Such a diagram is a graph of flow nodes (gateways, activities, events) connected by arcs (Fig.1).

Therefore we use standard graph-theoretic concepts like $\mathit{source}(\mathit{arc})$, $\mathit{target}(\mathit{arc})$ for source and target node of an *arc* (denoted *sourceRef* resp. *targetRef* and restricted by op.cit.Table 8.60 to certain flow nodes), $\mathit{pred}(\mathit{node})$ for the (possibly ordered) set of source nodes of arcs with target *node*, $\mathit{inArc}(\mathit{node})$ for the set of arcs ingoing the target *node*, $\mathit{succ}(\mathit{node})$ for the (possibly ordered) set of target nodes of arcs with source *node*, $\mathit{outArc}(\mathit{node})$ for the set of arcs outgoing the source *node*, etc. If in a diagram a node has only one incoming or one outgoing arc and if from the

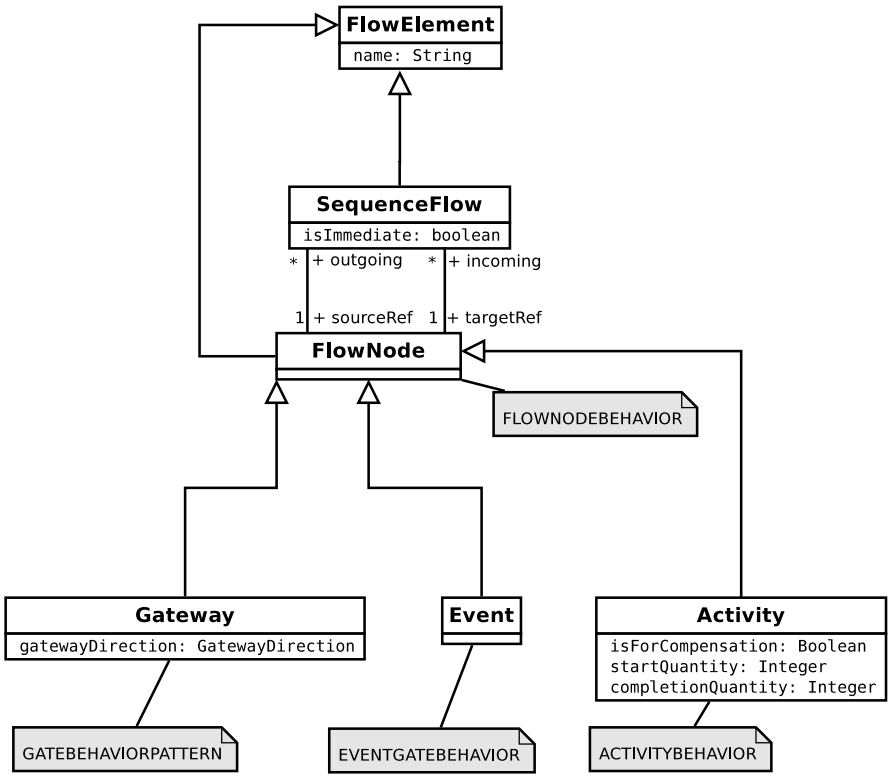

Fig. 1 Basic Class Hierarchy of Diagram Contents

context the node in question is clear, we write *in* / *out* instead of $inArc(node) = \{in\}$ $outArc(node) = \{out\}$.

We model the token-based BPMN interpretation of control flow by associating tokens—elements of an abstract set *Token*—to arcs, using a dynamic function $token(arc)$. Since a token is characterized by the process ID of the process instance *pi* to which it belongs (via its creation at the start of the process instance), we distinguish tokens belonging to different instances of one process *p*, writing $token_{pi}$ to represent the current token marking in the process diagram instance of the process instance *pi* a token belongs to. Thus $token_{pi}(arc)$ denotes the multiset of tokens belonging to process instance *pi* and currently residing on *arc*. We can suppress the parameter *pi* due to the single process view where *pi* is clear from the context.

For a *rule* at a target node of incoming arcs to become fireable some arcs must be *Enabled* by tokens being available at the arcs. This condition is usually required to be an atomic quantity formula stating that the number of tokens (belonging to a process instance *pi*) and currently associated to *in* (read: the cardinality of $token_{pi}(in)$, denoted $\mid token_{pi}(in) \mid$, used in particular in connection with complex gateways and called there *ActivationCount*, but also for readying activities where it is called

StartQuantity) is at least the quantity $inQty(in)$ required for incoming tokens at this arc. Unless otherwise stated the assumption is made that $inQty(in) = 1$, as suggested by the warning in op.cit.Table 10.3,Sect.14.2.2.

$Enabled(in) = (|\ token(in)\ | \geq inQty(in))$

Correspondingly the control operation CTLOP of a workflow usually consists of two parts, one describing which (how many) tokens are CONSUMEd on which incoming arcs and one describing which (how many) tokens are PRODUCEd on which outgoing arcs, indicated by using an analogous abstract function *outQty* (for activities called *CompletionQuantity*). We use macros to encapsulate the details.

CONSUME$(t, in) =$ DELETE$(t, inQty(in), token(in))$
PRODUCE$(t, out) =$ INSERT$(t, outQty(out), token(out))$
CONSUMEALL$(X) =$ **forall** $x \in X$ CONSUME(x)
PRODUCEALL$(Y) =$ **forall** $y \in Y$ PRODUCE(y)

The use of abstract DELETE and INSERT operations instead of directly updating $token(a,t)$ serves to make the macros usable in a concurrent context, where multiple agents may want to simultaneously operate on the tokens on an arc. It is also consistent with the special case that in a transition with both DELETE(in,t) and INSERT(out,t) one may have $in = out$, so that the two operations are not considered as inconsistent, but with their cumulative effect.

Structural relations between the consumed incoming and the produced outgoing tokens can be expressed by using an abstract function *firingToken*(A), which is assumed to select for each element a of an ordered set A of incoming arcs tokens from $token_{pi}(a)$ that enable a and can be CONSUMEd. *firingToken*$([a_1,\ldots,a_n]) = [t_1,\ldots,t_n]$ denotes that for each i, t_i is the (set of) token occurrence(s) selected to be fired on arc a_i. We write *firingToken*$(in) = t$ instead of *firingToken*$(\{in\}) = [t]$. Apparently the idea of a hierarchical token structure, which appeared in op.cit. and was modeled in [Börger and Thalheim(2008)], has been abandoned for BPMN 2.0 [Voelzer(2010b)] so that we write CONSUME(in) and PRODUCE(out) where the type of underlying tokens (assumed to belong to one process instance) is irrelevant or clear from the context.

2.3 Flow Nodes

The behaviorally central class is *FlowNode*, a subclass of *FlowElement* and coming with subclasses *Gateway*, *Activity*, *Event* (as explained above we disregard the fourth subclass *ChoreographyActivity*). Each instance *node* of this subclass represents a workflow construct whose behavioral meaning is expressed by a transition rule FLOWNODEBEHAVIOR$(node)$ stating upon which events and under which further conditions—typically on the control flow, the underlying data and the availability of resources—the rule can fire to execute the following actions:

- perform specific operations on the underlying data ('how to change the internal state') and control flow ('where to proceed'),
- possibly trigger new events (besides consuming the triggering ones) and releasing some resources.

FLOWNODEBEHAVIOR(*node*) =
  if *EventCond*(*node*) **and** *CtlCond*(*node*) **and** *DataCond*(*node*)
    and *ResourceCond*(*node*) **then**
      DATAOP(*node*)
      CTLOP(*node*)
      EVENTOP(*node*)
      RESOURCEOP(*node*)

FLOWNODEBEHAVIOR, associated with the class *FlowNode*, is a rule scheme, technically an ASM with well-defined semantics (see [Börger and Stärk(2003)]). Its abstractions are refined by further detailing in the next three sections the guards (conditions) respectively the operations (submachines) for workflow transitions to describe the behavioral meaning for instances of each of the three subclasses of *FlowNode*. When we need to consider to which process instance a flow *node* instance belongs we write *procInst*(*node*), to be distinguished from *process*(*node*) (the BPMN diagram) *node* belongs to.

3 Gateways

Gateway is a subclass of *FlowNode* used to describe the divergence (splitting) or convergence (merging) of control flow (op.cit.p.263) in two forms:

- to create parallel actions or to synchronize multiple actions,
- to select (one or more) among some alternative actions.

Gateway has five concrete subclasses for exclusive, inclusive, parallel, event-based and complex gateways (Fig.2), which come with specific constraints formulated in op.cit.Table 8.47 in terms of an attribute *gatewayDirection* on the number of their incoming and outgoing arcs.

Each gateway behavior is an instance of a scheme GATEBEHAVIORPATTERN associated with the abstract class *Gateway* and is defined as follows, refining the FLOWNODEBEHAVIOR: two (possibly ordered) sets of incoming respectively of outgoing arcs are selected where tokens are consumed respectively produced. To describe these sets we use functions $select_{Consume}(node)$ and $select_{Produce}(node)$ which will be constrained in various ways for specific gateways. The general control condition[2] is that all arcs in the selected (usually required or assumed to be non-empty) set of incoming arcs are enabled and that the process instance the gateway *node* belongs to is *Active* (see Sect. 4 for the concept of activity lifecycle). The control operation consists of a) consuming the firing tokens on each selected incoming arc and

[2] Except the special case analyzed in Sect. 3.4.1 of an event-based gateway used to start a process.

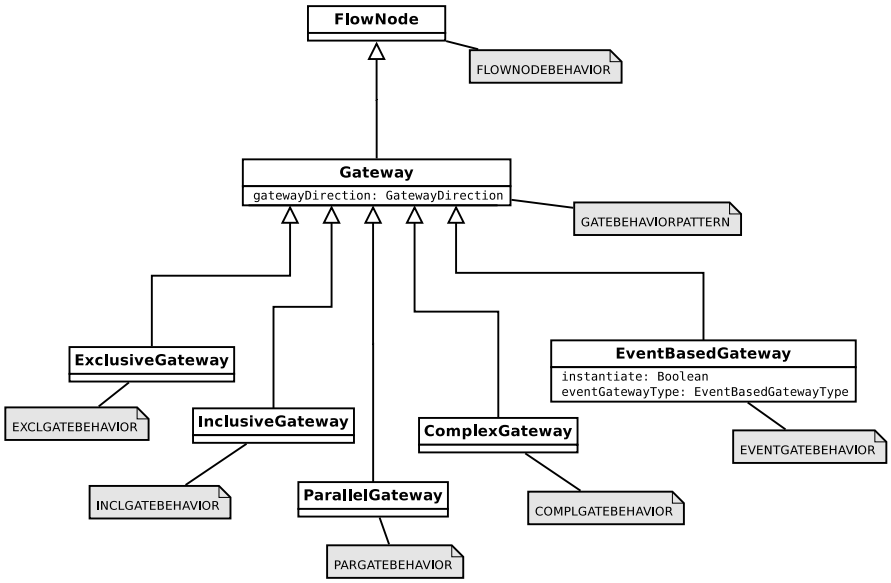

Fig. 2 Basic Class Hierarchy of Gateways

b) producing the required tokens on each selected outgoing arc (in the normal case that no exception does occur). DATAOP($node$) consists of multiple $assignments(o)$ associated to the outgoing arcs o.

The THROW($exc, node$) macro is used to indicate when an *exc*eption is thrown from a *node* to its possible catcher, triggering an event that is attached to the innermost enclosing scope instance (if any) and may be able to catch the *exc*eption. We assume a detailed definition of this macro to include the performance of the data association for throw events. This has the effect that when a throw event is triggered, this happens with the corresponding data in its scope assigned to what op.cit. calls the 'event data', from where the related catch event assigns them (see Sect. 5) to the so-called data elements in the scope of the catch event op.cit.10.4.1.

GATEBEHAVIORPATTERN($node$) =
  let $I = select_{Consume}(node)$
  let $O = select_{Produce}(node)$
    FLOWNODEBEHAVIOR($node, I, O$) **where**
$CtlCond(node, I) =$ **forall** $in \in I$ $Enabled(in)$ **and** $Active(procInst(node))$
CTLOP($node, I, O$) =
  CONSUMEALL($\{(t_j, in_j) \mid 1 \leq j \leq n\}$)
    where $[t_1, \ldots, t_n] = firingToken(I), [in_1, \ldots, in_n] = I$
  if $NormalCase(node)$ **then** PRODUCEALL(O)
    else THROW($GateExc, node$)
DATAOP($node, O$) = **forall** $o \in O$ **forall** $i \in assignments(o)$ ASSIGN(to_i,$from_i$)

Active(p)= (lifeCycle(p)=active)

We now refine this rule to the behavior of the five gateway subclasses.

3.1 Parallel Gateway (Fork and Join)

PARGATEBEHAVIOR is associated with the class *ParallelGateway*. Its behavior is to synchronize multiple concurrent branches (called AND-Join) by consuming one token on each incoming arc, and to spawn new concurrent threads (called AND-Split or Fork) by producing one token on each outgoing arc. A parallel gateway is not allowed to throw an exception. Thus it refines GATEBEHAVIORPATTERN.

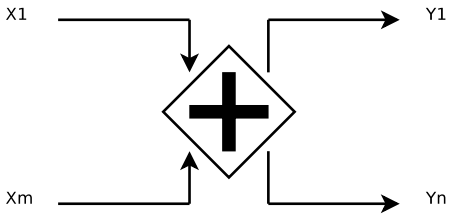

Fig. 3 Parallel Gateway – unconditionally spawn and synchronize threads of execution

PARGATEBEHAVIOR($node$) = GATEBEHAVIORPATTERN($node$) **where**
$select_{Consume}(node) = inArc(node)$ // AND-JOIN merging behavior
$select_{Produce}(node) = outArc(node)$ // AND-SPLIT (branching behavior
$NormalCase(node) = true$ // gate throws no exception
forall $in \in inArc(node)$ $inQty(in) = 1$
forall $out \in outArc(node)$ $outQty(out) = 1$[3]

3.2 Exclusive Gateway (Data-Based Exclusive Decision)

EXCLGATEBEHAVIOR is associated with class *ExclusiveGateway*.

Its behavior is to react to the enabledness of just one incoming arc (no matter which one, a feature named *pass-through semantics*), namely by consuming an enabling token, and to enable exactly one outgoing arc, namely the first one (in the diagram order) whose associated *DataCond*ition evaluates to true (so-called exclusive data-based decision). Usually a default case is specified to cover the situation where none of these *DataCond*itions is true; otherwise in this case an exception is thrown. Thus EXCLGATEBEHAVIOR is an instantiation of GATEBEHAVIORPATTERN.

[3] The two constraints on *inQty* and *outQty* seem to be intended for all flow node instances, except where state differently, so that from now on we assume them to be added implicitly.

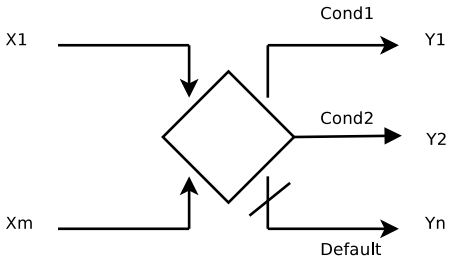

Fig. 4 Exclusive Gateway – choose exactly one thread of execution for synchronization and spawning

EXCLGATEBEHAVIOR($node$) = GATEBEHAVIORPATTERN($node$) **where**
| $select_{Consume}(node)$ |= 1 // exclusive merge
$select_{Produce}(node) = fst(\{a \in outArc(node) \mid DataCond(a)\})$
$NormalCase(node)$ if and only if
$\{a \in outArc(node) \mid DataCond(a)\} \neq \emptyset$ **or**
some default sequence flow is specified at *node*

3.3 Inclusive Gateway

INCLGATEBEHAVIOR is associated with class *InclusiveGateway*.

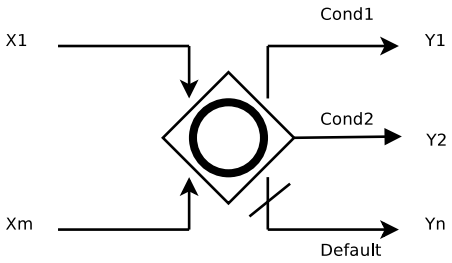

Fig. 5 Inclusive Gateway – synchronize and spawn some threads of execution

It enables every outgoing arc whose associated *DataCond*ition is true (branching [OmgBpmn(2009), 10.5.3]), with the same convention on exceptions as for exclusive gateways, and to synchronize the (required to be non-empty) set of incoming arcs which are enabled or have an "upstream token" ($UpstreamToken \neq \emptyset$) in the graph, not waiting for tokens on those unenabled arcs which "have no token upstream".

INCLGATEBEHAVIOR($node$) = GATEBEHAVIORPATTERN($node$) **where**
$select_{Consume}(node) =$ // NB. all to be enabled to fire
$\{in \in inArc(node) \mid Enabled(in)$ **or** $UpstreamToken(in) \neq \emptyset\}$
$select_{Produce}(node) = \{a \in outArc(node) \mid DataCond(a)\}$
$CtlCond(node, I, O) =$

$CtlCond_{\text{GATEBEHAVIORPATTERN}}(node, I, O)$ **and** $I \neq \emptyset$
$NormalCase(node) = NormalCase_{\text{EXCLGATEBEHAVIOR}}(node)$

An incoming arc "without token anywhere upstream" is defined in op.cit.Table 14.3 as unenabled arc to which there is no directed Sequence Flow *path* from any (arc with a) token unless

- *path* visits the inclusive gateway or
- *path* visits a node that has a directed path to a non-empty incoming sequence flow of the inclusive gateway and does not visit the gateway[4].

$t \in UpstreamToken(in)$ if and only if $InhibitingPath(t, in) \neq \emptyset$ **and**
thereIsNo $j \in inArc(node)$ $AntiInhibitingPath(t, j) \neq \emptyset$
where
$p \in InhibitingPath(t, in) =$
$p \in Path(t, in)$ **and** $token(in) = \emptyset$ **and** $target(in) \notin VisitedBy(p)$
$p \in AntiInhibitingPath(t, in) =$
$p \in Path(t, in)$ **and** $token(in) \neq \emptyset$ **and** $target(in) \notin VisitedBy(p)$
$VisitedBy(p) = \{n \mid n \in Node$ **and** n occurs as source or target on $p\}$
$Path(t, in) = Path(arc, in)$ **if** $t \in token(arc)$

3.4 Event-Based Gateway (Event-Based Exclusive Decision)

For event-based gateways the standard describes two behaviors, depending on whether the gateway is used to start a process or not, resulting in two ASM rules associated to the class *EventBasedGateway*. The EVENTGATEBEHAVIOR for the second case, in which the gateway is required to have some incoming sequence flow, is pictorially represented by Fig.6. In the first case the event-based gateway may have no (or only some special) incoming sequence flow; its EVENTGATEPROCSTARTBEHAVIOR is described in Sect.3.4.1.

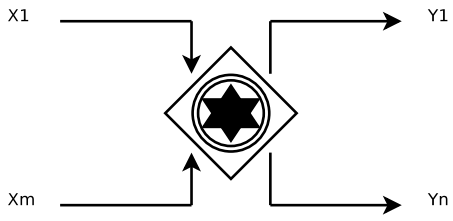

Fig. 6 Event-Based Gateway – choose exactly one thread of execution, depending on the first triggered gate event

EVENTGATEBEHAVIOR does not throw any exception. It has pass-through semantics for incoming sequence flow and the activated outgoing arc is defined to be

[4] The last conjunct has been added in [Voelzer(2010a)], correcting the definition which originally appeared in op.cit. Upstream tokens are called there tokens that have an inhibiting path but no anti-inhibiting path to the gateway.

the first one at which an associated *gateEvent Occurs* and can be CONSUMEd. Thus EVENTGATEBEHAVIOR refines GATEBEHAVIORPATTERN as follows:

- $select_{Consume}(node)$ chooses for each activation one incoming sequence flow,
- $select_{Produce}(node)$ yields one (dynamically determined[5]) outging sequence flow, namely the one whose associated *gateEvent Occurs* first (so-called exclusive event-based decision),
- $NormalCase(node) = true$: event-based gateways 'cannot throw any exception',
- the selected *gateEvent* is CONSUMEd.

We use a dynamic function *fst* to select an outgoing arc among those whose associated *gateEvent* (required to be either an *event* which has to be *Triggered* or a *receiveTask* which has to be *Completed*) *Occurs* 'first'; *fst* solves the conflict for concurrently occuring events. Receive tasks are tasks which wait for a message to arrive and are *Completed* by receiving the message (op.cit.p.139 and Sect.4.1).

$\text{EVENTGATEBEHAVIOR}(node) =$ // case with incoming arcs
$\quad \text{GATEBEHAVIORPATTERN}(node)$ **where**
$|\ select_{Consume}(node)\ | = 1$
$EventCond(node) =$ **forsome** $a \in outArc(node)\ Occurs(gateEvent(a))$
$select_{Produce}(node) = fst(\{a \in outArc(node) \mid Occurs(gateEvent(a))\})$
$\text{EVENTOP}(node) = \text{CONSUME}(gateEvent(select_{Produce}(node)))$
$NormalCase(node) =$ *true* // event gate throws no exception
$Occurs(gateEvent(a)) =$
$\quad \begin{cases} Triggered(event(a)) & \textbf{if } gateEvent(a) = event(a) \\ Completed(receiveTask(a)) & \textbf{if } gateEvent(a) = receiveTask(a) \end{cases}$

3.4.1 Event-Based Gateways for Process Start

If event-based gateways are used to start a process P, to be declared by setting their *instantiate* attribute to true, it is required that (except the case described in the next footnote) they have no incoming sequence flow—the only case of gateways with no ingoing arc op.cit.14.4.1.[6] The standard document considers two cases depending on whether there is only one event-based gateway (called *exclusive event-based gateway*) or a group of multiple event-based gateways which are used to start P. Such group elements are required to participate in the same conversation and at each gateway one event "needs to arrive; the first one creates

[5] The standard document interpretes this choice as "deferred until one of the subsequent Tasks or Events completes"op.cit.14.3.3. This creates an ambiguity for two successive enablings of the gate with deferred choice of an outgoing branch. We avoid this ambiguity by letting EVENTGATEBEHAVIOR only fire when the choice is possible due to at least one gate event occurring.

[6] The allowed case of incoming sequence flow whose source is an untyped start event (op.cit.p.276) is covered by the description explained below, including the usual conditions and operations for pass-through semantics.

a new process instance, while the subsequent ones are routed to the existing instance"(op.cit.14.4.1) "rather than creating new process instances" (op.cit.p.402). In both cases EVENTGATEPROCSTARTBEHAVIOR is obtained by analogous refinement conditions as for event-based gateways with incoming sequence flow, however the incoming arc selection and related control condition are empty and the control operation essentially creates a new instance of P.

To precisely reflect what is intended to happen when some expected gate events happen concurrently at multiple event-based gateways belonging to a same *group* (and to avoid a repetition for the first part of the behavior which is almost the same for singleton and multiple elements groups), we use a virtual node *group* to which EVENTGATEPROCSTARTBEHAVIOR is attached.[7] The formulation uses two *mode*s with corresponding subbehaviors, the second one being performed only if the group has more than one element. This reflects the requirement that for *group*s with multiple elements upon a 'first' event a new process instance is created "while the subsequent ones are routed to the existing instance"(op.cit.p.252).

EVENTGATEPROCSTARTBEHAVIOR($group$) =
  EVENTGATEPROCSTARTBEHAVIOR$_{Start}$($group$)
  EVENTGATEPROCSTARTBEHAVIOR$_{Progress}$($group$)

In $mode = Start$, EVENTGATEPROCSTARTBEHAVIOR$_{Start}$ upon the 'first' arrival of an event performs the following three actions:

- create a new instance of the to-be-started process and make it *Active*,[8]
- mimic the EVENTGATEBEHAVIOR(g) for a node $g \in group$ where a *gateEvent Occurs* 'first',
- in case there are other group members switch to $mode = Progress$, whereby the EVENTGATEPROCSTARTBEHAVIOR$_{Progress}$ becomes firable if some *gateEvent Occurs* at some other group member.

We use the dynamic abstract function *fst* here to select both a *group* member and an outgoing arc where a 'first' *gateEvent Occurs*.

EVENTGATEPROCSTARTBEHAVIOR$_{Start}$($group$) =
  GATEBEHAVIORPATTERN($group$) **where**
$select_{Consume}(group) = \emptyset$
$CtlCond(group) = (mode(group) = Start)$
$EventCond(group) =$ **forsome** $g \in group\ Occurs(gateEvent(g))$
let $g = fst(\{g \in group \mid Occurs(gateEvent(g))\})$
  $select_{Produce}(group) = fst(\{a \in outArc(g) \mid Occurs(gateEvent(g,a))\})$
    CTLOP($group, O$) =
      let $P =$ **new** $Instance(process(group))$

[7] The standard document makes the distinction in terms of an *eventGatewayType* attribute set to *parallel* for the case of multiple *group* elements.

[8] In general upon being enabled a process first becomes *Ready* and can GETACTIVE only after some input became available, see Sect. 4. But since at an event-based gateway the only allowed triggers are catch events or receive tasks which have no input data assignment, the newly created process becomes immediately *Active*.

PRODUCE($select_{Produce}(group)_P$)
$lastCreatedProcInst(group) := P$
$lifeCycle(P) := active$
$Seen(g) := true$
if $|\ group\ | > 1$ **then** $mode := Progress$
EVENTOP($group$) = CONSUME($gateEvent(select_{Produce}(g))$)
$NormalCase(group) =$ *true* // no event gate throws an exception
$Occurs(gateEvent(g)) =$ **forsome** $a \in outArc(g)$ $Occurs(gateEvent(g,a))$

EVENTGATEPROCSTARTBEHAVIOR$_{Progress}$ is executed in $mode = Progress$ each time a *gateEvent* for a remaining *group* member *Occurs*—until each *group* member has been *Seen*, in which case $mode(group)$ switches back to *Start* and resets *Seen*.[9] The standard document leaves this case underspecified. For definiteness we formulate here the following interpretation: a) once a *group* element has been *Seen* (because one of its *gateEvent*s *Occurs*), it is not reconsidered for another *gateEvent* to *Occur* before each group element has been *Seen*; b) no subsequent *gateEvent* PRODUCEs further tokens (on the arc where the *gateEvent Occurs*) before each group element has been *Seen*.

EVENTGATEPROCSTARTBEHAVIOR$_{Progress}(group) =$
GATEBEHAVIORPATTERN($group$) **where**
$select_{Consume}(group) = \emptyset$
$CtlCond(group) = (mode(group) = Progress)$
$EventCond(group) =$
forsome $g \in \{g \in group \mid \textbf{not}\ Seen(g)\}$ $Occurs(gateEvent(g))$
let $g = fst(\{g' \in group \mid Occurs(gateEvent(g'))\ \textbf{and not}\ Seen(g')\})$
$select_{Produce}(group) = fst(\{a \in outArc(g) \mid Occurs(gateEvent(g,a))\})$
EVENTOP($group$) = CONSUME($gateEvent(select_{Produce}(group))$)
CTLOP($group, O$) =
if $LastSeen(g, group)$ **then** // reset *group* state
$mode(group) := Start$
forall $g' \in group$ $Seen(g') := false$
else $Seen(g) := true$
PRODUCE($select_{Produce}(group)_{lastCreatedProcInst(group)}$)
$NormalCase(group) = true$
$LastSeen(g, group) = (group = \{g' \mid Seen(g')\} \cup \{g\})$

3.5 Complex Gateway

COMPLGATEBEHAVIOR is associated with class *ComplexGateway*. It has two rules (op.cit.Table 14.5): COMPLGATEBEHAVIOR$_{start}$ describing the behavior in mode *waitingForStart* and COMPLGATEBEHAVIOR$_{reset}$ for *reset* mode.

[9] This reflects the standard document requirement that "one event out of each group" has to arrive to complete the process instance created upon the 'first' arrival of an event.

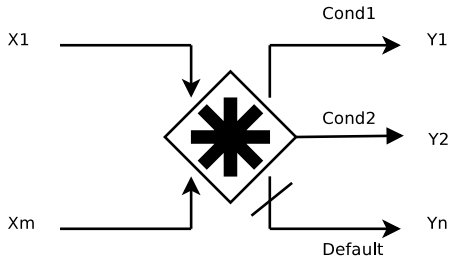

Fig. 7 Complex Gateway – user-defined splitting/synchronizing behaviour

$$\text{COMPLGATEBEHAVIOR} = \begin{array}{l}\text{COMPLGATEBEHAVIOR}_{start}\\ \text{COMPLGATEBEHAVIOR}_{reset}\end{array}$$

If *waitingForStart*, a complex gateway waits for its *activationCondition* to become true. This attribute expresses a (somehow restricted) condition on data and the number of tokens on incoming arcs (called *activationCount*) so that we represent it as *DataCond*. When the rule fires, it consumes a token from each enabled incoming arc and produces a token on each outgoing arc whose associated condition is true. The evaluated expressions may depend on the value of *waitingForStart*. If none of these conditions is true and no default flow has been specified, an exception is thrown. In addition, when no exception occured a) the mode switches by setting *waitingForStart* to false and b) the set of in *waitingForStart* mode enabled incoming arcs (where therefore a token has been consumed) is recorded for use in *reset* mode. Thus $\text{COMPLGATEBEHAVIOR}_{start}$ refines GATEBEHAVIORPATTERN as follows:

$\text{COMPLGATEBEHAVIOR}_{start}(node) = \text{GATEBEHAVIORPATTERN}(node)$ **where**
$DataCond(node) = activationCondition(node)$ **and** $waitingForStart(node)$
$select_{Consume}(node) = \{in \in inArc(node) \mid Enabled(in)\}$
$select_{Produce}(node) = \{o \in outArc(node) \mid DataCond(a) = true\}$
$\text{CTLOP}(node, I, O) =$
  $\text{CTLOP}_{\text{GATEBEHAVIORPATTERN}}(node, I, O)$
  if $NormalCase(node)$ **then**
    $atStartEnabledArc(node) := select_{Consume}(node)$
    $waitingForStart := false$
$NormalCase(node) = NormalCase_{\text{EXCLGATEBEHAVIOR}}(node)$

In the *reset* case (i.e. if $waitingForStart = false$), a complex gateway awaits a token on each incoming arc that has not been enabled when *waitingForStart*, except on not enabled arcs which have no token upstream (as defined above for inclusive gateways). It consumes tokens from each of these arcs, produces a token on each outgoing arc whose associated condition is true and resets its mode to $waitingForStart = true$. No exception is thrown in *reset* mode. Thus $\text{COMPLGATEBEHAVIOR}_{reset}$ is an instantiation of GATEBEHAVIORPATTERN, refined as follows.

$\text{COMPLGATEBEHAVIOR}_{reset}(node) = \text{GATEBEHAVIORPATTERN}(node)$ **where**
  $DataCond(node) =$ **not** $waitingForStart(node)$
  $select_{Consume}(node) = \{in \in inArc(node) \setminus atStartEnabledArc(node) \mid$

$Enabled(in)$ **or** $UpstreamToken(in) \neq \emptyset\}$ // NB. all to be enabled to fire
$select_{Produce}(node) = \{o \in outArc(node) \mid DataCond(a) = true\}$
CTLOP$(node, I, O) =$
CTLOP$_{\text{GATEBEHAVIORPATTERN}}(node, I, O)$
$waitingForStart := true$
$NormalCase(node) = true$ // no exception thrown in mode *reset*

4 Activities

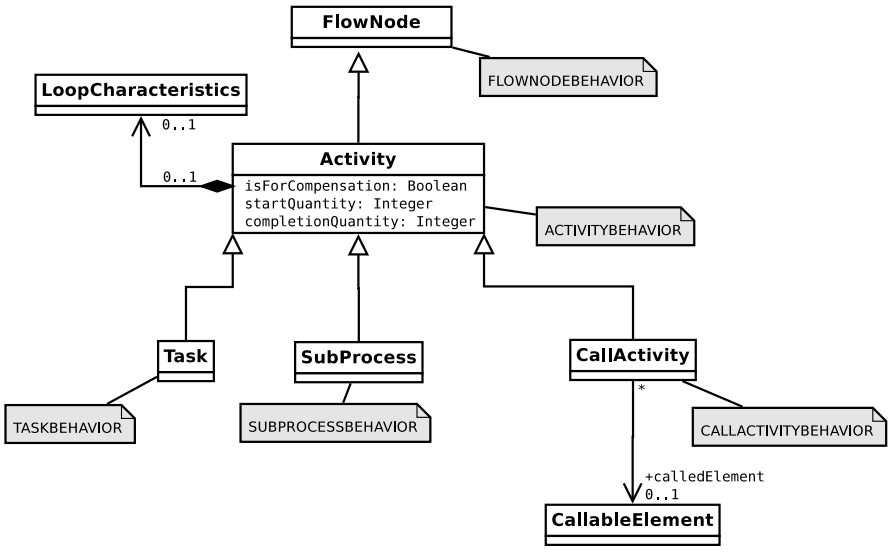

Fig. 8 Basic Class Hierarchy of Activities

The *Activity* subclass of *FlowNode* is associated with an ACTIVITYBEHAVIOR which describes the general form of the behavior of an *Activity* node, whether atomic or compound and whether performed once or repeatedly. It is refined for each of the three subclasses *Task*, *SubProcess* and *CallActivity* of *Activity* (see Fig.8).

Activities have associated *InputSets* and *OutputSets* which define the data requirements for input/output to/from the activity (via an *InputOutputSpecification* op.cit.Fig.10.54). At least one InputSet must be *Available* for the activity to become *Active* with data input from the first *Available* set; at the *completion* of the activity, some data output may be produced from the first *Available* OutputSet if it satisfies the activity's IORule expressing a relationship between that OutputSet and the InputSet used to start the activity. An exception is thrown if there is no *Available* OutputSet at all or if the IORule is not satisfied for the first *Available* OutputSet (op.cit.Sect.14.2.2).

Activities can be without incoming sequence flow. Examples are compensation activities, (event) subprocesses, a Receive task in a process without start event which is used to start this process. We treat such specialized activities separately and describe first the behavior of activities which do have some incoming sequence flow.

When at least one among possibly multiple incoming arcs is *Enabled*[10] (so-called uncontrolled flow) a new activity instance is created, is linked by a *parent* function to the process that triggered it and becomes *Ready* waiting to GETACTIVE.[11] The two parameters for the corresponding set *InstSet* of process instances and the trigger process *TriggerProc* will be instantiated below to describe variations of this ACTIVITYENTRY behavior. If the activity is not *Interrupted*, its lifecycle switches to *Active*, to STARTEXECution, after an input set from *inputSets* became *Available* (in which case this set is recorded for use when the activity is completed) [OmgBpmn(2009), p.130,393]. $select_{InputSets}$ expresses which input set is chosen, specified by the standard document as the first available input set (with respect to a given order). INTERRUPT among others switches the *lifeCycle* to *Withdrawn*, *Failed* or *Terminated*; we abstain from completing here the loose declaration of intents for the activity lifecylce in op.cit.14.2.2.

ACTIVITYENTRY($node, InstSet, TriggerProc$) = FLOWNODEBEHAVIOR($node$)
where
$CtlCond(node)$ = **forsome** $in \in inArc(node)$ $Enabled(in)$
CTLOP($node$) =
  let $arc = select_{Consume}(\{in \in inArc(node) \mid Enabled(in)\})$
    CONSUME($firingToken(arc), arc$)
  let a = **new** *InstSet*
    $lifeCycle(a) := ready$
    $parent(a) := TriggerProc$
  step GETACTIVE($a, node$)[12]
GETACTIVE($a, node$) =
  if $Ready(a)$ **and forsome** $i \in inputSets(node)$ $Available(i)$ **then**
    let $i = select_{InputSets}(\{i \in inputSets(node) \mid Available(i)\})$
      STARTEXEC($a, node$)
      $lifeCycle(a) := active$
      $currInputSet(node) := i$
  if $Interrupted(a)$ **then** INTERRUPT(a)
$Ready(a) = (lifeCycle(a) = ready)$

[10] Enabledness is defined here to mean that "the required number of *Tokens* ... *StartQuantity* ... is available", as reflected by our macro definition in Sect. 2.2.

[11] The 1.0 standard version required in addition that the activity has no currently active instances, in accordance with the suggested transformation to BPEL. Such an additional guard guarantees that all instances of an activity are ultimately triggered by one enabling token, which reflects the intended termination behavior of all activity instances in case of a failure. Probably also for 2.0 this guard should be added.

[12] **step** denotes the interruptable FSM-like variant of sequential execution of ASMs (see [Börger and Craig(2009)] for an explicit definition).

ACTIVITYBEHAVIOR is an instance of ACTIVITYENTRY where the instance *node* of the activity is added to the set of instances of this activity in the process instance *node* belongs to and the *parent* process is this process instance.

ACTIVITYBEHAVIOR(*node*) =
  ACTIVITYENTRY(*node*, *Instance*(*node*, *procInst*(*node*)), *procInst*(*node*))

In the following subsections this rule is instantiated for the three *Activity* subtypes by refining the abstract STARTEXEC machine. See Sect. 4.4 for the instantiation for iterated activities (standard loops and multi-instance loops).

4.1 Tasks

A task is an atomic activity describing work in the given process that "cannnot be broken down to a finer level of detail"(op.cit.Sect.10.2.3), although it may take its (in the process not traceable) execution time. This atomicity is expressed by the sequentiality operator **seq** for structuring ASMs (see [Börger and Stärk(2003), Ch.4]), which turns a low-level sequential execution view of two machines *M* followed by *N* into a high-level atomic view of one machine *M* **seq** *N*.

Therefore STARTEXEC(*task*, *t*) means to a) EXECute the *task* (instance to which the triggering token *t* belongs), whose exact definition depends on the type of the *task*, and b) when the execution is *Completed* without failure to produce outgoing sequence flow (*CompletionQuantity*(*task*) many tokens on each arc outging the *task* op.cit.p.130,393) possibly together with some output.[13] $select_{OutputSets}$ is defined as yielding the first available output set in a given order op.cit.p.393. Thus TASKBEHAVIOR refines ACTIVITYBEHAVIOR as follows.

TASKBEHAVIOR(*node*) = ACTIVITYBEHAVIOR(*node*) **where**
  STARTEXEC(*a*, *node*) = EXEC(*a*) **seq**
    if *Completed*(*a*) **then** EXIT(*a*, *node*)
    if *Interrupted*(*a*) **then** INTERRUPT(*a*)
    if *CompensationOccurs*(*a*) **then**
      TRIGGERCOMPENSATION(*a*)
      lifeCycle(*a*) := *compensating*
  EXIT(*a*, *node*) =
    forall *o* ∈ *outArc*(*node*) PRODUCE(*o*)[14]
    DELETE(*a*, *Instance*(*node*, *procInst*(*node*)))
    PUSHOUTPUT(*a*, *node*)
  PUSHOUTPUT(*a*, *node*) =

[13] We skip the cases that a task may fail or terminate due to a fault in the environment. We also skip the *Completing* activity mode, which is forseen for the final 2.0 version of the standard but not yet furthermore specified in op.cit.p.393.

[14] Here again our macro definition of PRODUCE captures that the "number of tokens indicated by ... *CompletionQuantity* is placed" on the outgoing arcs, see op.cit.pg.393.

if forall $o \in outputSets(node)$ **not** $Available(o)$
then THROW$(noAvailOutputExc, node)$
else let $o = select_{OutputSets}(\{o \in outputSets(node) \mid Available(o)\})$
if $IORules(node)(o, currInputSet(a)) = false$
then THROW$(noIORulesExc, node)$
else PUSH$(output(o))$

Remark. In the case of an activity without outgoing sequence flow, **forall** $o \in outArc(task)$ PRODUCE(o) is an empty rule so that if there are no end events in the containing (sub)process the activity terminates here.

There are seven types (subclasses) of *Task*, each coming with characteristic attributes, constraints and meaning of EXECution (op.cit.Fig.10.10); not furthermore specified tasks are considered as abstract tasks.

$$TaskType = \{Send, Receive, Service, User, Manual, Script, BusinessRule\}$$

Each of these subclasses is associated with a refinement of TASKBEHAVIOR defined by refining EXEC$(task[,i])$ and $Completed(task)$ as follows. A further specification of the abstractions we use in these definitions appears either in the standard document or comes with the task instantiation. For example, RECEIVE(m) is described as 'waiting for m until it arrives' [OmgBpmn(2009), 14.2.3], $job(t)$ for $type(t) \in \{Service, Script, User, Manual\}$ as the associated service or script or user task or manual task (also denoted $operationRef(t)$ for service tasks). Since abstract tasks (read: with undefined type) are considered as 'never actually executed by an IT system', we treat them here as empty actions.

EXEC$(t) =$ **let** $i = currInputSet(a)$ **in**

$$\begin{cases} \text{SEND}(payload(mssg(t)), receiver(mssg(t))) & \textbf{if } type(t) = Send \\ \text{RECEIVE}(mssg(t)) & \textbf{if } type(t) = Receive \\ \text{INVOKE}(job(t), i) & \textbf{if } type(t) \in \{Service, Script\} \\ \text{ASSIGN}(job(t), i, performer(job(t), i)) & \textbf{if } type(t) \in \{User, Manual\} \\ \text{CALL}(businessRule(t), i) & \textbf{if } type(t) = BusinessRule \\ \textbf{skip} & \textbf{if } Abstract(t) \end{cases}$$

$Sent(mssg(t))$ is described for t of type *Send* as true 'upon instantiation' of t, $Received(mssg(t))$ for t of type *Receive* as true'when the message arrives'.

$Completed(t) =$

$$\begin{cases} Sent(mssg(t)) & \textbf{if } type(t) = Send \\ Received(mssg(t)) & \textbf{if } type(t) = Receive \\ Completed(job(t)) & \textbf{if } type(t) \in \{Service, Script\} \\ Completed(businessRule(t)) & \textbf{if } type(t) = BusinessRule \\ Done(job(t)) & \textbf{if } type(t) \in \{User, Manual\} \\ true & \textbf{if } Abstract(t) \end{cases}$$

There is a special case which requires an additional rule. A Receive *task* which is "used to start a Process", a fact indicated by an $Instantiate(task)$ flag, is required to either have no *in*coming arc in its associated process without start event, or to

have an *in*coming arc with $source(in)$ being a start event of the associated process (op.cit.p.139). For the first case a special instance of FLOWNODEBEHAVIOR($task$) is added which has no control condition and no control operation and where $EventCond(task)$ is defined as $Received(mssg(task))$.

There are also further refinement constraints for some tasks. For example service tasks are required to have exactly one input set and at most one output set.

4.2 Sub-Processes

Subprocesses are activities which encapsulate a process (op.cit.p.394). They define a contextual *scope* that can be used for attribute visibility, for the handling of transactions, events, exceptions and compensations (op.cit.p.152). Their behavior concerning exception handling and compensation is described below when explaining the behavior of intermediate events that are placed on the boundary of an activity. Their normal behavior along their inner sequence flow is described by the behavior of tasks, events and gateways which constitute their internal details. What remains to be described for arbitrary subprocesses is a) the activation of subprocesses, which involves an activity instantiation and passing data from caller to callee, and b) how to EXIT subprocesses upon their completion. For the special internal control and exit behavior of elements of the *AdHocProcess* subclass of class *SubProcess* see Sect. 4.2.2, for elements of the subclass *Transaction* of *SubProcess* Sect. 4.2.3.

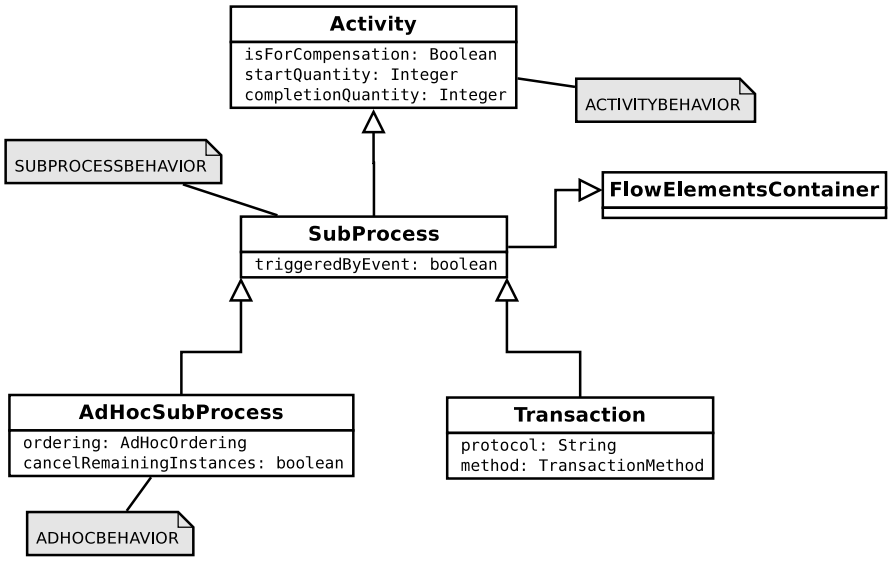

Fig. 9 Basic Class Hierarchy of Subprocesses

4.2.1 Subprocess Activation

There are two cases of subprocess activation, depending on whether the subprocess *node* has incoming sequence flow or not. In the first case the associated SUBPROCESSBEHAVIOR refines ACTIVITYBEHAVIOR, in the second case SUBPROCESSNOINFLOWBEHAVIOR refines FLOWNODEBEHAVIOR.

For a subprocess with some incoming sequence flow its activation is triggered through tokens produced by the *caller* process on the (unique) incoming arc. It consists in a) creating a new instance of the subprocess as child process of the caller process (which we retrieve, for the sake of example, from the token produced by the latter on the arc *in*coming the subprocess) and b) triggering its start.

Triggering the new process instance has in [OmgBpmn(2009), 14.2.4] two versions, depending on whether the subprocess either has a (and then unique) *startEvent* or otherwise a non-empty set *StartNode* of "activities and gateways without incoming sequence flow". In the first subcase simply the *startEvent* is triggered.

In the second subcase we interpret the stipulation that "all such activities and gateways get a *token*" by associating in the graph with each $n \in StartNode$ a (virtual) entry arc $in(n)$ which can be enabled by producing a new token on it (in the new process instance, thus triggering n there; using a process subscript distinguishes elements in the current process from their analogues in the new instance).

SUBPROCESSBEHAVIOR($node$) = ACTIVITYBEHAVIOR($node$) **where**
STARTEXEC(a, $node$) =
  if $startEvent(node) \neq$ **undef then**
    let $\{t\} = trigger(startEvent(a))$
    $TriggerOccurs_P(t, startEvent(a)) := true$
  else
    forall $n \in StartNode(node)$ PRODUCE($startToken(a, node), in(n)$)

For a subprocess without incoming sequence flow, it is required that there is a non-empty set *StartEvent* of "start events that are target of sequence flow from outside the sub-process" [OmgBpmn(2009), 14.2.4]. It is stipulated that each such start event "that is reached by a *token*"[15] generates a new subprocess instance, similar to the pass-through semantics for incoming sequence flow. In other words a triggered start event with a trigger is chosen, it is consumed, a new process instance is created and the trigger of the chosen start event is set in the new subprocess instance.

For the special case of a so-called *event subprocess* (denoted by setting the *triggeredByEvent* flag) it is required that it has no incoming and no outgoing sequence flow and exactly one *startEvent*, so that $StartEvent = \{startEvent\}$. In this case a new instance is started each time this *startEvent* is triggered while the *parent*

[15] This sounds ambiguous: where should the token arrive if there is no incoming sequence flow? We interpret it as meaning that some *caller* process triggers a start event in the callee, the *targetyRef* subprocess node [OmgBpmn(2009), p.215].

process is active.[16] The *parent* process can be interrupted or not, depending on whether the start event *isInterrupting* or not.

We incorporate both behaviors in one refinement EVENTSUBPROCESSBEHAVIOR of FLOWNODEBEHAVIOR, defined for event subprocess *node*s as follows. The corresponding start event rule defined in Sect. 5 describes how the new subprocess instance starts its execution once (one of) its start event is triggered.

EVENTSUBPROCESSBEHAVIOR($node$) = FLOWNODEBEHAVIOR($node$)
where
  $EventCond(node) =$
    forsome $e \in StartEvent(node)$ $Happened(e)$
      and if $triggeredByEvent(node)$ **then** $Active(parent(procInst(node)))$
  let $e = select_{StartEvent}(\{n \in StartEvent(node) \mid Happened(e)\})$
  let $\{t\} = select_{Trigger}\{t \in trigger(e) \mid TriggerOccurs(t,e)\}$
    EVENTOP($node$) = CONSUME(t,e)
    CTLOP($node$) =
      let $P =$ **new** $Instance(process(node))$
        $caller(P) := \begin{cases} parent(procInst(node)) & \textbf{if } triggeredByEvent(node) \\ caller(node) & \textbf{else} \end{cases}$
        $TriggerOccurs_P(t,e) := true$
      if $isInterrupting(node)$ **then** CANCEL($parent(procInst(node))$)
  $Happened(e) =$ **forsome** $t \in trigger(e)$ $TriggerOccurs(t,e)$

4.2.2 Ad-hoc Processes

Ad-hoc processes are called non-operational elements for which "only conceptual model is provided which does not specify details neede to execute them on an engine" op.cit.p.389. This means that the standard document intends to specify ad-hoc processes only loosely so that we leave their treatment here at the same degree of underspecification. Each subprocess marked as ad-hoc has a static set of *InnerActivities* intended to be executed (if *Enabled*) in an order that is mainly "determined by the performers of the activities" op.cit.10.2.5,p.161. We denote by $EnabledInnerAct(node)$ the runtime set of *Enabled* elements of *InnerActivities*, which is required to be initially the set of inner activities without incoming sequence flow (op.cit.14.2.5). We reflect the performers' choice by a function $select_{EnabledInnerAct(node)}$ together with a monitored predicate *ActivationTime* to express the moment where a new selection takes place. Nevertheless an *adHocOrdering* function is provided to specify either a parallel execution (the default case that the dynamic and initially empty set *RunningInnerAct* of concurrently running inner activities is finite) or a sequential execution (where "only one activity

[16] op.cit.p.156. Sect.14.4.4 says "Running", a term which is not defined in Fig.14.2. and seems to request an active parent process only for initiating a non-interrupting event subprocess. We disregard here the baroque looking additional feature mentioned on p.405 that "An Event Sub-Process can optionally retrigger the Event through which it was triggered, to cause its continuation outside the boundary of the associated Sub-Process."

can be performed at a time"(op.cit.Table 10.22) so that *RunningInnerAct* is empty or a singleton set). An *AdHocCompletionCond*ition is evaluated each time an inner activity completes and defines whether the subprocess completes by EXITing (possibly producing some output). In the parallel case this depends on whether the attribute *CancelRemainingInstances* is true: if it is, all elements of *RunningInnerAct* are CANCELed, otherwise the ad-hoc subprocess is required to wait for completion until each element of *RunningInnerAct* has completed or terminated. We use the **await** *Cond M* construct to describe such waiting for the execution of *M* until the *Cond*ition becomes true, as defined for ASMs in [Altenhofen and Börger(2009)].

Therefore the behavior ADHOCBEHAVIOR of class *AdHocProcess* elements is the following refinement of ACTIVITYBEHAVIOR. For simplicity of exposition and without loss of generality we assume that each launched inner activity upon its completion enables a (virtual) link which enters the evaluation of the *AdHocCompletionCond*ition of its ad-hoc subprocess.

ADHOCBEHAVIOR(*node*) = ACTIVITYBEHAVIOR(*node*) **where**
STARTEXEC(*a*, *node*) =
  while not *AdHocCompletionCond*(*node*)
    if *adHocOrdering*(*node*) = *Sequential* **then** LAUNCHINNERACT(*node*)
    if *adHocOrdering*(*node*) = *Parallel* **then**
      if *ActivationTime*(*node*) **then** LAUNCHINNERACT(*node*)
  seq
    if *CancelRemainingInstances*(*node*) **then**
      forall *a* ∈ *RunningInnerAct*(*node*)
        CANCEL(*a*)
        EXIT(*a*, *node*)
    else awaitforall *a* ∈ *RunningInnerAct*(*node*)
      Completed(*a*) **or** *Terminated*(*a*)
        EXIT(*node*)
LAUNCHINNERACT(*node*) =
  if *enabledInnerAct*(*node*) ≠ ∅ **then**
    let *e* = $select_{EnabledInnerAct(node)}$(*EnabledInnerAct*(*node*))
      ACTIVITYBEHAVIOR(*e*)
      INSERT(*e*, *RunningInnerAct*(*node*))
      DELETE(*e*, *EnabledInnerAct*(*node*))

4.2.3 Transaction

Transactions are subprocesses whose behavior is also controlled by a transaction protocol, which is assumed to be given. They come with a special method to undo a transaction when it is cancelled. The behavioral instantiation of a transaction comes up to add in the specification of the entry and exit actions the details for creating the transactional scope and for what should happen when a transaction fails (rollback and possibly compensation of the involved processes). We do not specify this behavior here because it is only loosely hinted at in the BPMN standard document.

4.3 Call Activity

Any *CallActivity* (also called reusable sub-process) "calls a pre-defined process" and "results in the transfer of control" to the "CallableElement being invoked", using the data inputs and outputs as well as InputSets and OutputSets of the *referenced* callable element [OmgBpmn(2009), 10.2.5/6]. We denote the called activity by $activity(reference(node))$, of which a new instance is created and added to the set of active instances of the activity, having triggered one of its start events (possibly provided with some available input).

CALLACTIVITYBEHAVIOR($node$) =
  ACTIVITYENTRY($node, Instance(activity(reference(node))), node$)
where STARTEXEC($a, node$) =
  choose $n \in \{n \in StartEvent(a) \mid trigger(n) = None\}$
  $TriggerOccurs_a(None, n) := true$
  INSERT($a, ActiveProcInst(activity(reference(node)))$)

4.4 Iterated (Loop) Activities

Loop and Multiple Instances activities act as wrapper for an activity that can be iterated respectively spawn multiple instances in parallel or sequentially. We interprete the wrapper as providing the input, but probably other interpretations are allowed by the standard. An activity with *LoopCharacteristics* has an iterative behavior either of *StandardLoopCharacteristics* or of *MultiInstanceLoopCharacteristics* type (op.cit.Fig.10.6).

The standard loop characteristics defines a *LoopCond*ition which is checked, as indicated by a *testBefore* attribute, either before or after an execution of the loop *body* to decide whether the loop completes at this moment or not:

- If *testBefore* is true, then *LoopCond* is evaluated before the first iteration of the to be iterated activity is started (and then again after each iteration), in which case the loop activity corresponds to the **while** construct,
- If *testBefore* is false, then *LoopCond* is evaluated *after* the first iteration has finished (and then again after each iteration), in which case the loop activity corresponds to the **until** construct.

A *loopMaximum* can be used in the *loopCond*. We use a function *inputs* to describe the data flushed to the selected input set $currInputSet(node)$ in the following refinement STANDARDLOOPBEHAVIOR of ACTIVITYBEHAVIOR. To ACTIVATE the loop body means to trigger the execution of the BPMN process defined by the body; ACTIVATE is defined depending on the type of its argument process. For simplicity of exposition and without loss of generality we make a similar assumption as for the ADHOCBEHAVIOR rule, namely that each body process upon its completion enables a (virtual) link which enters the evaluation of the *loopCond*ition.

STANDARDLOOPBEHAVIOR($node$) = ACTIVITYBEHAVIOR($node$) **where**
  STARTEXEC(a,$node$) =
    let i = *inputs*(*currInputSet*($node$))
      if *testBefore*($node$) = *true* **then**
        while *loopCond*(a,$node$) ACTIVATE(*body*(a,$node$),i)
      if *testBefore*($node$) = *false* **then**
        until *loopCond*($node$) ACTIVATE(*body*(a,$node$),i)
    seq if *Completed*(a,$node$) **then** EXIT(a,$node$)
  Completed(a,$node$) =
    $\begin{cases} \textbf{not } \mathit{loopCond}(a,node) & \textbf{if } \mathit{testBefore}(node) = \mathit{true} \\ \mathit{loopCond}(a,node) & \textbf{if } \mathit{testBefore}(node) = \mathit{false} \end{cases}$

The multi-instance loop characteristics determines how many instances of an activity are spawned to be executed sequentially or in parallel. A *loopCardinality* expression defines the number of to be created activity instances and an attribute *isSequential* determines whether the instances are executed sequentially ("a new instance is generated only after the previous one has been completed") or in parallel. As for ad-hoc activities, a *MiCompletionCond*ition is evaluated each time an instance completes and when it becomes true, the remaining instances are cancelled and the multi-instance loop completes. There are four types of instance completion *behavior* determining "when events shall be thrown from an activity instance that is about to complete" [OmgBpmn(2009), Table 10.26]:

- Case *behavior* = *All*: "no event is ever thrown; a token is produced after completion of all instances".
- Case *behavior* = *None*: An event *noneBehaviorEventRef* is thrown each time an instance completes.
- Case *behavior* = *One*: An event *oneBehaviorEventRef* is thrown "upon the first instance completing" .
- Case *behavior* = *Complex*: a *complexBehaviorDefinition* determines "when and which events are thrown".

MULTINSTLOOPBEHAVIOR refines ACTIVITYBEHAVIOR in two respects:

- Refining the input selection and output production to data collections whose elements are associated to the activity instances; this is a signature refinement defined in op.cit.Sect.14.2.7, as is the corresponding refinement of the PUSHOUTPUT(p) component of EXIT(p) for multiple instance activities p.
- Refining the definition of STARTEXEC.

For simplicity of exposition and without loss of generality we make a similar assumption as for the STANDARDLOOPBEHAVIOR rule, namely that each inner activity instance upon its completion enables a (virtual) link entering the *MiCompletionCond*ition evaluation. The events thrown by EMITEVENT each time an inner activity completes are instances of the class *ImplicitThrowEvent*, read: events that are automatically thrown to be caught by a boundary event on the multi-instance activity (op.cit. Table 10.28). The standard document does not explain the data input/output behavior of multiple instances, so that we do not enter its formalization here.

MULTINSTLOOPBEHAVIOR = ACTIVITYBEHAVIOR(*node*) **where**
STARTEXEC(*a*,*node*) =
  while *MiCompletionCond*(*a*,*node*) = *false*
    if *isSequential*(*node*) **then**
      LAUNCHINSTANCE(*node*) // run first instance until completion
      step // creation of further instances
        while *loopCardinality*(*node*) >| *ActiveInnerAct*(*a*,*node*) |
          LAUNCHINSTANCE(*a*,*node*) // run next instance until completion
    else // parallel case: new instances created at activation time
      while *loopCardinality*(*node*) >| *ActiveInnerAct*(*a*,*node*) |
        if *ActivationTime*(*node*) **then** // run one more instance
          LAUNCHINSTANCE(*a*,*node*)
  step
    forall *b* ∈ *ActiveInnerAct*(*a*,*node*) CANCEL(*b*)
    EXIT(*a*,*node*) // NB with refined PUSHOUTPUT
LAUNCHINSTANCE(*a*,*n*) =
  let *act* = **new** *Instance*(*innerAct*(*n*))
    { ACTIVATE(*act*)
    { INSERT(*act*,*ActiveInnerAct*(*a*,*n*))
  step await *Completed*(*act*) EMITEVENT(*n*)
EMITEVENT(*n*) =
  { THROW(*noneBehaviorEventRef*(*n*),*n*) **if** *behavior*(*n*) = *None*
  { THROW(*oneBehaviorEventRef*(*n*),*n*) **if** *behavior*(*n*) = *One*
                                and | *Instance*(*innerAct*(*n*)) |= 1
  { **forall** *e* ∈ *ComplexBehaviorDefinition*
      THROW(*e*,*n*) **if** *behavior*(*n*) = *Complex*

5 Events

Events are used in BPMN to control the execution order or timing of process activities (op.cit.8.3.6). *Event* splits into two subclasses *ThrowEvent* and *CatchEvent* both of which can contain intermediate events, which may throw or catch triggers, the causes of events. *EndEvent*s are *ThrowEvent*s because they typically "throw" a result when a process ends, whereas *StartEvent*s "catch" a trigger to start a process and thus form a subclass of *CatchEvent*, as do the elements of *BoundaryEvent* which are typically attached as intermediate events to an activity. When an event is thrown, its trigger is propagated to the innermost enclosing scope instance where an attached event can catch the trigger. For some cases (e.g. for errors or escalations) it is intentionally left underspecified what should happen when no catching event can be found.

We indicate by *trigger*(*node*) the set of types of event *trigger*s that may be associated to *node* as defined in op.cit.Table 10.77: a message (arriving from another

participant), a timer, a condition, a signal (broadcasted from another process) or none (and in event subprocesses also escalation, error or compensation).

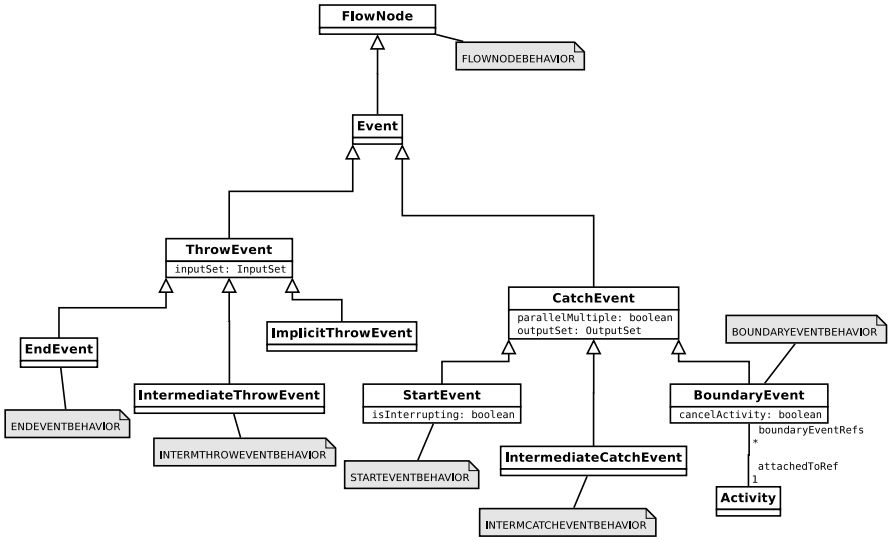

Fig. 10 Basic Class Hierarchy of Events

In the following subsections we explain the behavior of these *Event* subclasses, mostly abstracting from the data events may carry. The (throw) behavior of so-called implicit throw events, which are used in connection with multi-instance activities, has already been described when defining the EMITEVENT macro in Sect. 4.4.

5.1 Start Events

Fig. 11 Start Events – None, Message, Timer, Escalation, Error, Compensation, Signal, Multiple, Parallel Multiple

A start event has no incoming arc (except when attached to the boundary of a subprocess to which a higher-level process may connect) and every process con-

taining some (possibly more than one) start event is required to have no other flow elements without incoming sequence flow (except intermediate events attached to an activity boundary, event subprocesses or compensation activities, see below) [OmgBpmn(2009), 10.4.2]. When at a start event a *TriggerOccurs*—a predicate representing that an event "happens" during the course of a business process, see a definition in Sect. 5.3.2—a new process instance is created and started by producing a (unique) *startToken* on every outgoing arc.

If there are multiple ways to trigger a process only one trigger is required to occur except in the special case where all elements of $trigger(node)$ must be triggered to instantiate the process. This is expressed by the following two refinements of FLOWNODEBEHAVIOR($node$) for start event *node*s without incoming arc.[17]

STARTEVENTBEHAVIOR($node$) = FLOWNODEBEHAVIOR($node$)
  where // normal case without parallel multiple trigger
    $EventCond(node) = ParallelMultiple \notin trigger(node)$ **and**
      forsome $e \in trigger(node)$ $TriggerOccurs(e, node)$
    EVENTOP($node$) =
      choose $e \in \{e \in trigger(node) \mid TriggerOccurs(e, node)\}$
        CONSUME($triggerOccurence(e)$)
    CTLOP($node$) =
      let $P =$ **new** $Instance(process(node))$
      forall $o \in outArc_P(node_P)$ PRODUCE($startToken_P(node), o$)

STARTEVENTPARMULTBEHAVIOR($node$) = FLOWNODEBEHAVIOR($node$)
  where // case with parallel multiple triggers
    $EventCond(node) = ParallelMultiple \in trigger(node)$ **and**
      forall $e \in trigger(node) \setminus \{ParallelMultiple\}$
        $TriggerOccurs(e, node)$
    EVENTOP($node$) =
      forall $e \in trigger(node) \setminus \{ParallelMultiple\}$
        CONSUME($triggerOccurence(e)$)
    CTLOP($node$) =
      let $P =$ **new** $Instance(process(node))$
      forall $o \in outArc_P(node_P)$ PRODUCE($startToken_P(node), o$)

In the special case of a start event *node* with an incoming arc *in*, the event is required to be attached to the boundary of a subprocess to which a higher-level process may connect. This situation can be modeled by treating *in* as a virtual incoming arc which can be *Enabled* by a token produced by the higher-level process, so that $Triggered(node)$ is instantiated to $Enabled(in(node))$ and CONSUMEVENT($node$) to CONSUME($firingToken(in(node)), in(node)$).

Remark on processes without a start event. Constructs without incoming arc and belonging to a process without a start event are required to be activated when

[17] Since in this chapter we do not investigate BPMN choreography features, we disregard the case of start events which participate in a conversation including other start events where only one new process instances is created for the specific conversation.

the process is instantiated. For simplicity of exposition we model such processes by equipping them with a virtual start event from which a virtual arc leads to each construct without incoming sequence flow. Then these constructs are all triggered when by the instantiation of the process the start event is triggered.

Table 10.77 op.cit. explains how *TriggerOccurs* is defined. For a conditional trigger e CONSUME($triggerOccurence(e)$) is required to let the corresponding condition become false between two occurrences of that trigger.

5.2 End Events

Fig. 12 End Events – None, Message, Escalation, Error, Cancel, Compensation, Signal, Multiple, Termination

An end event is used to indicate where a process will end and thus has incoming arcs (where each arriving token will be consumed) and no outgoing sequence flow (except when the end event is attached to the boundary of a subprocess from where a higher-level process may proceed); furthermore every process containing some (possibly more than one) end event is required to have no other flow elements without outgoing sequence flow (except compensation activities, see below) (op.cit.10.4.3). An end event may emit (possibly multiple) results belonging to its *resultType* set containing elements of the following types: message, signal, terminate, error, escalation, cancel, compensation or none (op.cit.Table 10.81]).

Thus ENDEVENTBEHAVIOR refines FLOWNODEBEHAVIOR as follows.

ENDEVENTBEHAVIOR($node$) = FLOWNODEBEHAVIOR($node$) **where**
$CtlCond(node)$ = **forsome** $in \in inArc(node)$ $Enabled(in)$
CTLOP($node$) = **choose** $in \in \{in \in inArc(node) \mid Enabled(in)\}$
  CONSUME($firingToken(in), in$)
  if $Multiple \notin resultType(node)$ // normal case without multiple results
  then let $\{res\} = resultType(node)$ **in** EMITRESULT($res, node$)
  else forall $res \in resultType(node) \setminus \{Multiple\}$
    EMITRESULT($res, node$)

EMITRESULT is detailed in op.cit.Table 10.81. For message result type the *MessageFlow* determines the message(s) to be sent from the *sender* to the *receiver*. For signal result type the corresponding *signalRef* is BROADCAST from *node* to

'any process that can receive it' (we write $receivers(signalRef(node), node)$). Error result type yields THROWing an error—to be caught by an enclosing intermediate event if there is any, otherwise it is intentionally left unspecified what should happen—and terminating all the activities which are currently active in the subprocess (assumed to include all instances of multi-instances). Similarly THROWing an *Escalation* or *Cancel* type from *node* has the effect to trigger an enclosing $targetIntermEv(escalation, node)$ in an attempt to catch the escalation and in case it is not caught there to propagate it further up; the cancel case is required to be used only within a transaction subprocess, with $targetIntermEv(resType, node)$ attached to the boundary of the transaction, and in addition a transaction protocol cancel message has to be sent to any entities involved in the transaction; we represent this behavior as a CALLBACK to each participant in the set $listener(Cancel, node)$. Compensation result type yields THROWing a compensation event, which is required to activate the compensation handler of the corresponding activity (or set of activities) $actRef(node)$ after their completion. If $resType = Terminate$ 'all activities in the process should be immediately ended', including multiple instances; this can be achieved by deleting all tokens on any arc in the given $process(node)$ and in any active inner activity, deleting the latter from the set of active inner activities. For $resType = None$ in the special case of an end *node* of a subprocess which completed, when the subprocess is *Completed* the flow has to go back to the caller process, to which effect a token is PRODUCEd on the arc *out*going the caller of the process instance to which the end *node* belongs and the process instance is deleted from the set of active instances of the called $activity(reference(node))$.

EMITRESULT$(resType, node) =$
  if $resType = Message$ **then forall** $m \in MessageFlow$
    if $sender(m) = node$ **then** SEND$(payload(m), receiver(m))$
  if $resType = Signal$ **then**
    BROADCAST$(signalRef(node), receivers(signalRef(node), node))$
  if $resType = Error$ **then**
    THROW$(error, node)$
    forall $a \in ActiveActivity(process(node))$ TERMINATE(a)
  if $resType \in \{Cancel, Escalation\}$ **then**
    THROW$(resType, node)$
    if $resType = Cancel$ **then**
      CALLBACK$(mssg(Cancel, node), listener(Cancel, node))$
  if $resType = Compensation$ **then**
    THROW$((compensation, actRef(node)), node)$
  if $resType = Terminate$ **then** INTERRUPT$(process(node))$
  if $resType = None$ **and** $IsSubprocessEnd(node)$
    and $Completed(process(node))$ **then**
      PRODUCE$(returnToken(node), out(caller(process(node))))$
      DELETE$(process(node), ActiveProcInst(activity(reference(node))))$
  where
    CALLBACK$(m, L) =$ **forall** $l \in L$ SEND$(payload(m), l)$

INTERRUPT(p) =
  DELETEALLTOKENS(p)
  forall $q \in$ *ActiveInnerAct*(p)
    DELETEALLTOKENS(q)
    DELETE(q,*ActiveInnerAct*(p))

5.3 Intermediate Events

Intermediate events occur between start and end events and may either throw or catch triggers, namely to send or receive messages or to establish a condition or to react to its satisfaction, where the conditions may concern timing features or exceptions or compensations. If an intermediate event is enabled during normal process flow, it will either ("throw use") immediately set off the event trigger and perform its normal sequence flow CTLOP (CONSUME its enabling token and PRODUCE tokens on its outgoing sequence flow) or ("catch use") wait to perform its normal CTLOP until its trigger occurs. When intermediate events are used in an *activity* to describe exception or compensation handling which is outside the normal flow of the *activity*, they are attached to the boundary of that *activity* (represented by *attachedTo* = *activity*), formally as elements of *boundaryEventRefs*(*activity*). Such events can only catch their triggers during an execution of the activity they are attached to, thereby starting an exception or compensation flow which may interrupt the activity (as error or cancel intermediate events always do).

The intermediate events which can be used in normal flow or as attached to an activity boundary are listed in [OmgBpmn(2009), Tables 10.82/3]. In the following two sections we describe the associated normal flow behavior; for the boundary event behavior see Sect. 5.4.

5.3.1 Intermediate Throw Events in Normal Flow

Fig. 13 Intermediate Throw Events – Message, Escalation, Compensation, Signal, Multiple

An intermediate throw event is required to have some ('uncontrolled' if multiple) incoming and (except intermediate link events) some (simultaneously activated if multiple) outgoing sequence flow (op.cit.10.4.4). The details of its event operation SETEVENTTRIGGER depend on the trigger type associated to the event.

SETEVENTTRIGGER yields message SENDing in case of a *Message* trigger type, a BROADCAST for *Signal* trigger type, triggering (the unique) *targetLink* for trigger type *Link* and THROWing an escalation or compensation[18] for *Escalation* or *Compensation* trigger type. If $trigger(node)$ contains multiple trigger elements, then SETEVENTTRIGGER$(node,t)$ is performed for each trigger element $t \in trigger(node)$.

Thus INTERMEDIATETHROWEVENTBEHAVIOR refines FLOWNODEBEHAVIOR as follows and is associated to the class INTERMEDIATETHROWEVENT.

INTERMEDIATETHROWEVENTBEHAVIOR$(node)$ =
  FLOWNODEBEHAVIOR$(node)$ **where**
$CtlCond(node)$ = **forsome** $in \in inArc(node)$ $Enabled(in)$
CTLOP$(node)$ = **choose** $in \in \{in \in inArc(node) \mid Enabled(in)\}$
  CONSUME$(firingToken(in), in)$
  PRODUCEALL$(outArc(node))$[19]
EVENTOP$(node)$ =
  if $Multiple \notin trigger(node)$ // case with only one trigger
    then let $\{t\} = trigger(node)$ **in** SETEVENTTRIGGER$(t, node)$
    else forall $t \in trigger(node) \setminus \{Multiple\}$ SETEVENTTRIGGER$(t, node)$
SETEVENTTRIGGER(t,n) =

$$\begin{cases} \textbf{forall } m \in MessageFlow \textbf{ with } sender(m) = node & \\ \quad \text{SEND}(payload(m), receiver(m)) & \textbf{if } t = Message \\ \text{BROADCAST}(signalRef(n), receivers(signalRef(n), n)) & \textbf{if } t = Signal \\ Triggered(targetLink(n)) := true & \textbf{if } t = Link \\ \text{THROW}(escalation, n) & \textbf{if } t = Escalation \\ \text{THROW}((compensation, actRef(node)), node) & \textbf{if } t = Compensation \end{cases}$$

5.3.2 Intermediate Catch Events in Normal Flow

Fig. 14 Intermediate Catch Events – Message, Timer, Escalation, Error, Cancel, Compensation, Signal, Multiple, Parallel Multiple

[18] We do not describe further details about compensation because this concept is only unsatisfactorily sketched in the standard document, in particular when it comes to speak about compensation of multiple activities.

[19] If for a source intermediate link event $outArc(node) = \emptyset$, then PRODUCEALL$(\emptyset)$ = *SKIP*.

An intermediate catch event, when token *Enabled*, will wait to perform its normal CTLOP until its *EventCond*ition is satisfied expressing that the triggers to be caught occur. When it becomes true the normal CTLOPeration is performed and the occuring event triggers are consumed (where relevant, e.g. for link type where the *Triggered* predicate at *sourceLink*(*node*) has to be reset to false), op.cit.10.4.6.

Thus INTERMEDIATECATCHEVENTBEHAVIOR refines FLOWNODEBEHAVIOR as follows and is associated to the class INTERMEDIATECATCHEVENT. The predicate *TriggerOccurs*$(t, node)$ is defined in op.cit.Table 10.82.

INTERMEDIATECATCHEVENTBEHAVIOR$(node)$ =
  FLOWNODEBEHAVIOR$(node)$ **where**
CtlCond$(node)$ = **forsome** $in \in inArc(node)$ *Enabled*(in)
EventCond$(node)$ =
    (*ParallelMultiple* $\notin$ *trigger*$(node)$ // only one trigger required to occur
    and forsome $t \in trigger(node)$ *TriggerOccurs*$(t, node)$)
  or
    (*ParallelMultiple* $\in$ *trigger*$(node)$ // all triggers required to occur
      and forall $t \in trigger(node)$ *TriggerOccurs*$(t, node)$)
EVENTOP$(node)$ =
  let *TriggOcc* $= \{t \in trigger(node) \mid TriggerOccurs(t, node)\}$
    if *ParallelMultiple* $\notin$ *trigger*$(node)$ **then**
      choose $t \in TriggOcc$ CONSUME(*triggerOccurence*(t))
    else forall $t \in TriggOcc$ CONSUME(*triggerOccurence*(t))
CTLOP$(node)$ = **choose** $in \in \{in \in inArc(node) \mid Enabled(in)\}$
  CONSUME(*firingToken*$(in), in$)
  PRODUCEALL(*outArc*$(node)$)
TriggerOccurs$(t, node)$ =

$$\begin{cases} \textbf{forsome } m \in \textit{Message Received}(m, node) & \textbf{if } t = \textit{Message} \\ \textit{TimerCondition}(node) = \textit{true} & \textbf{if } t = \textit{Timer} \\ \textit{EventExpression}(node) = \textit{true} & \textbf{if } t = \textit{Conditional} \\ \textit{Triggered}(\textit{sourceLink}(node)) = \textit{true} & \textbf{if } t = \textit{Link} \end{cases}$$

As [OmgBpmn(2009), Table 10.94] shows, the *TimerCondition*$(node)$ typically involves *timeData*$(node)$ or *cycleData*$(node)$. Timer as well as conditional triggers are implicitly thrown, meaning that when activated they wait until *TriggerOccurs*, namely when their time based or state based condition becomes true.

5.4 Boundary Events

An intermediate event that is *attachedTo* the boundary of an *ac*tivity has no incoming but has (possibly multiple) outgoing sequence flow—except intermediate events with a *Compensation* trigger which are required not to have any outgoing sequence

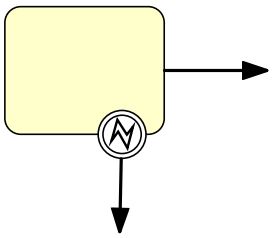

Fig. 15 A Task Activity with an Intermediate Catching Error Event attached to its boundary

flow, although they may have an outgoing association. When a boundary intermediate event is triggered, three things happen: a) the event trigger occurence is consumed; b) if the *cancelActivity*(*act*) attribute is true,[20] the *act*ivity is INTERRUPTed (including all its inner activity instances in case of a multi-instance *act*ivity, see the definition in Sect. 5.2); c) the CTLOP enables the outgoing sequence flow activating an event handler [OmgBpmn(2009), p.234,253,14.4.3]. For a compensation event trigger to occur means that the *toBeCompensatedAct*ivity has *Completed*, so that the compensation handler for that activity is activated (for which reason a compensation event is required to be non interrupting).

Thus BOUNDARYEVENTBEHAVIOR refines FLOWNODEBEHAVIOR. The definition of *TriggerOccurs*(*t*,*node*) from Sect. 5.3.2 is extended by op.cit.Table 10.83.

BOUNDARYEVENTBEHAVIOR(*node*) = FLOWNODEBEHAVIOR(*node*) **where**
EventCond(*node*) =
  (*ParallelMultiple* $\notin$ *trigger*(*node*) // only one trigger required to occur
  and forsome $t \in$ *trigger*(*node*) *TriggerOccurs*(*t*,*node*))
or
  (*ParallelMultiple* $\in$ *trigger*(*node*) // all triggers required to occur
  and forall $t \in$ *trigger*(*node*) *TriggerOccurs*(*t*,*node*))
EVENTOP(*node*) =
let *TriggOcc* = $\{t \in$ *trigger*(*node*) $\mid$ *TriggerOccurs*(*t*,*node*)$\}$
if *ParallelMultiple* $\notin$ *trigger*(*node*) **then choose** $t \in$ *TriggOcc*
CONSUME(*triggerOccurence*(*t*))
if $t =$ *Compensate* **then**
ACTIVATE(*compensation*(*attachedTo*(*node*)))
else forall $t \in$ *TriggOcc*
CONSUME(*triggerOccurence*(*t*))
if *Compensate* $\in$ *TriggOcc* **then**
ACTIVATE(*compensation*(*attachedTo*(*node*)))
CTLOP(*node*) =
PRODUCEALL(*outArc*(*node*))
if *cancelActivity*(*attachedTo*(*node*)) **then** INTERRUPT(*attachedTo*(*node*))

[20] It is required to always hold for Error and to never hold for Compensate type.

$$
\begin{array}{l}
\mathit{TriggerOccurs}(t,\mathit{node}) = \\
\quad \left\{ \begin{array}{ll}
\textbf{forsome } m \in \mathit{Message\ Received}(m,\mathit{node}) & \textbf{if } t = \mathit{Message} \\
\mathit{TimerCondition}(\mathit{node}) = \mathit{true} & \textbf{if } t = \mathit{Timer} \\
\mathit{EventExpression}(\mathit{node}) = \mathit{true} & \textbf{if } t = \mathit{Conditional} \\
\textbf{forsome } n\ \mathit{node} \in \mathit{receivers}(\mathit{signalRef}(n),n) & \\
\quad \textbf{and } \mathit{Arrived}(\mathit{signalRef}(n),\mathit{node}) & \textbf{if } t = \mathit{Signal} \\
\mathit{triggerOccurence}(t) = (\mathit{Completed},a) \textbf{ and } \mathit{Completed}(a) & \textbf{if } t = \mathit{Compensate} \\
\mathit{Caught}(t,\mathit{node}) & \\
\quad \textbf{if } t \in \{\mathit{Escalation},\mathit{Error},\mathit{Cancel}\} &
\end{array} \right.
\end{array}
$$

6 An Example

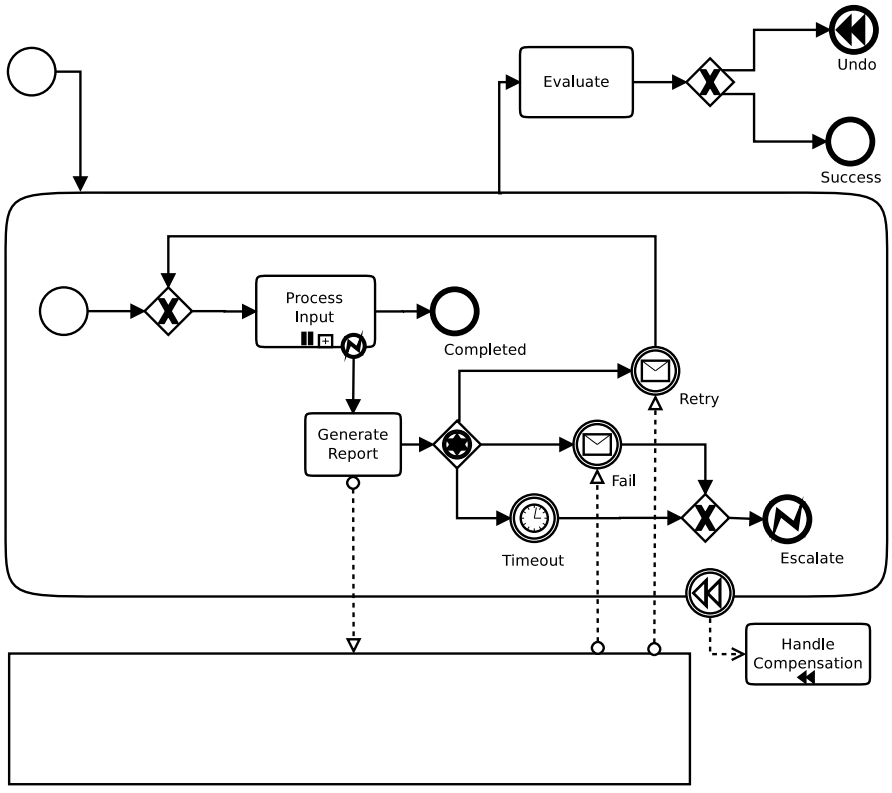

Fig. 16 Example – A compensatable process with a remote arbiter

We illustrate the preceding definitions by the workflow in Fig.16. It has two pools: one is used as an abstract blackbox for an external participant and is left

empty, the other one is assumed to contain all the other elements and is drawn with an invisible outline.

The workflow execution begins with the Start Event in the upper left corner. Since there is no specific trigger type associated with this event, it has to be triggered *manually*. When such a manual *TriggerOccurs*, *EventCond* is true and the underlying BPMN scheduler can choose to fire this Start Event. This process consumes through the EVENTOP the event trigger and produces through CTLOP a token on the outgoing edge, thus enabling the follwing subprocess to start. The subprocess is instantiated by SUBPROCESSBEHAVIOR through triggering its start event.

Within the subprocess, the exclusive gateway is used as a join. It can fire upon the arrival of the produced single incoming token because EXCLGATEBEHAVIOR restricts its consumption function by $|select_{Consume}(node)| = 1$. The next workflow element is a collapsed subprocess "Process Input" which loops over some not furthermore specified input, using multiple instances, as indicated by the vertical bars. After consuming the firing token through an application of ACTIVITYBEHAVIOR, the refined version of STARTEXEC for Multi Instance Looping Activities invokes its inner activity with simultaneous multiple instances because the loop is marked as non-sequential. The exact mechanism involves mapping a collection of input data to those instances by means of some Input Association and is carried out by the interpreter. The inner process activities are instantiated via the ACTIVATE macro.

In case none of the created instances raises an error, the loop activity will eventually be finalised via EXIT, which places a token on the ougoing edge of the loop activity. In case at least one instance of the loop subprocess raises an error that is not caught within its scope, all instances of the subprocess are terminated by EMITRESULT. The interpreter searches now for a suitable boundary event matching this error, rethrowing it on every level on the process hierarchy. In the example there is a catching intermediate event of type error directly on the boundary of the "Process Input" subprocess. Assuming that its definition matches the error that was thrown, the interpreter will signal its associated trigger, thus fulfilling its *EventCond*.

The only flow node which can be fired at this stage is the "Generate Report" task. Apart from collecting and processing information about the caught error, it dispatches a message to the external participant represented by the empty pool. From the local point of view of the single example process no details about what happens in the empty pool are known except that its participant adheres to the message-exchange indicated in the diagram. The following event-based gateway due to its *EventCond* can only be enabled after one of its associated events is triggered. This can either be an external message requesting to repeat the "Process Input" activity, or a message informing that the process failed, or a failure to receive at least one of these messages within a certain timeframe. The latter case is also regarded as a failure. Choosing the appropriate outgoing edge to place a token on relies on an ordering among the triggers, as expressed by the *fst* function that is used in $select_{Produce}$ within EVENTGATEBEHAVIOUR.

In the *Retry* case, a token is placed on the edge leading back to the exclusive join, resulting in another iteration. In the failure and timeout cases, the token is produced on a path that ends with the Error end event "Escalate". The intermediate

event on whose incoming edge the token is placed initially has in all possible cases a true *EventCond*ition and thus can fire without delay. This is guaranteed by the $select_{Produce}$ function of the event-based gateway and the fact that the triggers are only cleared by the following events.

The subprocess can be exited via one of the two end events or via some uncaught terminating trigger further down in the process hierarchy. The "Escalate" end event signals a terminal failure of the process and throws an error that can possibly be caught in an enclosing context to handle the exception. Because the outermost process that is modeled in this diagram has no designated event handlers, it would be automatically be terminated as well when this end event is triggered. The other end event does not throw any other than the none trigger, so if the subprocess token arrives there it is *Completed* and thus via EMITRESULT returns to the caller process.

Finally, the toplevel process exits via either the "Success" or the "Undo" end event. In the first case, control is returned to the calling instance higher in the hierarchy, or the interpereter if there was no such instance. The "Undo" end event has an attached trigger of type compensation. The trigger that it throws contains a reference to the activity that is to be compensated, as expressed by the *actRef* function in EMITRESULT. Compensations are different from errors in that they are usually directed from an outer towards an inner context. As described in the text, we assume that the interpreter will catch the trigger and find the associated compensation activity of the enclosed reference. In our case, the "Undo" end event references the inner subprocess, so the "Handle Compensation" activity would be invoked to undo the effects of the "Process Input" subprocess.

7 Conclusion

One could simplify considerably the BPMN execution semantics by restricting it to a core of BPMN constructs in terms of which all the other constructs can be defined, to streamline the standard as suggested already in [Börger and Thalheim(2008)]. Up to now missing or ambiguous issues one would like to see clarified by the standard document can be integrated into the model once they are decided. This holds in particular for a satisfactory specification of the lifecycle concept for activities and its relation with exception handling and compensation.

Acknowledgements We thank Wei Wei and Son Thai for helpful comments on the first draft of this chapter and Hagen Völzer for information on the current status of the work in the OMG BPMN standardization committee.

Draft of April 10. Final version to appear in: D. Embley and B. Thalheim (Eds): Handbook of conceptual modelling. Springer-Verlag, 2010.

Appendix: BPMN in a Nutshell

We list here the behavioral rules associated with the subclasses of the BPMN *FlowNode* class.

FLOWNODEBEHAVIOR($node$) =
  if $EventCond(node)$ **and** $CtlCond(node)$ **and** $DataCond(node)$
    and $ResourceCond(node)$ **then**
      DATAOP($node$)
      CTLOP($node$)
      EVENTOP($node$)
      RESOURCEOP($node$)

7.1 GateWay Behavior

GATEBEHAVIORPATTERN($node$) =
  let $I = select_{Consume}(node)$
  let $O = select_{Produce}(node)$
    FLOWNODEBEHAVIOR($node, I, O$)
where
  $CtlCond(node, I)$ = **forall** $in \in I$ $Enabled(in)$ **and** $Active(procInst(node))$
  CTLOP($node, I, O$) =
    CONSUMEALL($\{(t_j, in_j) \mid 1 \leq j \leq n\}$) **where**
      $[t_1, \ldots, t_n] = firingToken(I), [in_1, \ldots, in_n] = I$
    if $NormalCase(node)$ **then** PRODUCEALL(O)
      else THROW($GateExc, node$)
  DATAOP($node, O$) = **forall** $o \in O$ **forall** $i \in assignments(o)$ ASSIGN($to_i, from_i$)
  $Active(p) = (lifeCycle(p) = active)$

PARGATEBEHAVIOR($node$) = GATEBEHAVIORPATTERN($node$) **where**
  $select_{Consume}(node) = inArc(node)$ // AND-JOIN merging behavior
  $select_{Produce}(node) = outArc(node)$ // AND-SPLIT (branching behavior
  $NormalCase(node) = true$ // gate throws no exception

EXCLGATEBEHAVIOR($node$) = GATEBEHAVIORPATTERN($node$) **where**
  $| select_{Consume}(node) | = 1$ // exclusive merge
  $select_{Produce}(node) = fst(\{a \in outArc(node) \mid DataCond(a)\})$
  $NormalCase(node) = NormalCase_{\text{EXCLGATEBEHAVIOR}}(node)$

INCLGATEBEHAVIOR($node$) = GATEBEHAVIORPATTERN($node$) **where**
  $select_{Consume}(node)$ = // NB. all to be enabled to fire
    $\{in \in inArc(node) \mid Enabled(in)$ **or** $UpstreamToken(in) \neq \emptyset\}$
  $select_{Produce}(node) = \{a \in outArc(node) \mid DataCond(a)\}$
  $CtlCond(node, I, O)$ =

$CtlCond_{\text{GATEBEHAVIORPATTERN}}(node, I, O)$ **and** $I \neq \emptyset$
$NormalCase(node)$ if and only if // as for the exclusive case
$\{a \in outArc(node) \mid DataCond(a)\} \neq \emptyset$ **or**
some default sequence flow is specified at *node*

EVENTGATEBEHAVIOR$(node) =$ // case with incoming arcs
GATEBEHAVIORPATTERN$(node)$ **where**
$|\, select_{Consume}(node) \,| = 1$
$EventCond(node) =$ **forsome** $a \in outArc(node)$ $Occurs(gateEvent(a))$
$select_{Produce}(node) = fst(\{a \in outArc(node) \mid Occurs(gateEvent(a))\})$
EVENTOP$(node) =$ CONSUME$(gateEvent(select_{Produce}(node)))$
$NormalCase(node) =$ *true* // event gate throws no exception
$Occurs(gateEvent(a)) =$
$\begin{cases} Triggered(event(a)) & \textbf{if } gateEvent(a) = event(a) \\ Completed(receiveTask(a)) & \textbf{if } gateEvent(a) = receiveTask(a) \end{cases}$

EVENTGATEPROCSTARTBEHAVIOR$(group) =$
EVENTGATEPROCSTARTBEHAVIOR$_{Start}(group)$
EVENTGATEPROCSTARTBEHAVIOR$_{Progress}(group)$

EVENTGATEPROCSTARTBEHAVIOR$_{Start}(group) =$
GATEBEHAVIORPATTERN$(group)$ **where**
$select_{Consume}(group) = \emptyset$
$CtlCond(group) = (mode(group) = Start)$
$EventCond(group) =$ **forsome** $g \in group$ $Occurs(gateEvent(g))$
let $g = fst(\{g \in group \mid Occurs(gateEvent(g))\})$
$select_{Produce}(group) = fst(\{a \in outArc(g) \mid Occurs(gateEvent(g,a))\})$
CTLOP$(group, O) =$
let $P =$ **new** $Instance(process(group))$
PRODUCE$(select_{Produce}(group)_P)$
$lastCreatedProcInst(group) := P$
$lifeCycle(P) := active$
$Seen(g) := true$
if $|\, group \,| > 1$ **then** $mode := Progress$
EVENTOP$(group) =$ CONSUME$(gateEvent(select_{Produce}(g)))$
$NormalCase(group) =$ *true* // no event gate throws an exception
$Occurs(gateEvent(g)) =$ **forsome** $a \in outArc(g)$ $Occurs(gateEvent(g,a))$

EVENTGATEPROCSTARTBEHAVIOR$_{Progress}(group) =$
GATEBEHAVIORPATTERN$(group)$ **where**
$select_{Consume}(group) = \emptyset$
$CtlCond(group) = (mode(group) = Progress)$
$EventCond(group) =$
forsome $g \in \{g \in group \mid$ **not** $Seen(g)\}$ $Occurs(gateEvent(g))$
let $g = fst(\{g' \in group \mid Occurs(gateEvent(g'))$ **and not** $Seen(g')\})$
$select_{Produce}(group) = fst(\{a \in outArc(g) \mid Occurs(gateEvent(g,a))\})$

EVENTOP(*group*) = CONSUME(*gateEvent*(*select*$_{Produce}$(*group*)))
CTLOP(*group*, *O*) =
if *LastSeen*(*g*, *group*) **then** // reset *group* state
mode(*group*) := *Start*
forall $g' \in group$ *Seen*(g') := *false*
else *Seen*(*g*) := *true*
PRODUCE(*select*$_{Produce}$(*group*)$_{lastCreatedProcInst(group)}$)
NormalCase(*group*) = *true*
LastSeen(*g*, *group*) = (*group* = $\{g' \mid Seen(g')\} \cup \{g\}$)

$$\text{COMPLGATEBEHAVIOR} = \begin{array}{l}\text{COMPLGATEBEHAVIOR}_{start} \\ \text{COMPLGATEBEHAVIOR}_{reset}\end{array}$$

COMPLGATEBEHAVIOR$_{start}$(*node*) = GATEBEHAVIORPATTERN(*node*) **where**
DataCond(*node*) = *activationCondition*(*node*) **and** *waitingForStart*(*node*)
select$_{Consume}$(*node*) = $\{in \in inArc(node) \mid Enabled(in)\}$
select$_{Produce}$(*node*) = $\{o \in outArc(node) \mid DataCond(a) = true\}$
CTLOP(*node*, *I*, *O*) =
CTLOP$_{\text{GATEBEHAVIORPATTERN}}$*node*, *I*, *O*)
if *NormalCase*(*node*) **then**
atStartEnabledArc(*node*) := *select*$_{Consume}$(*node*)
waitingForStart := *false*
NormalCase(*node*) = *NormalCase*$_{\text{EXCLGATEBEHAVIOR}}$(*node*)

COMPLGATEBEHAVIOR$_{reset}$(*node*) = GATEBEHAVIORPATTERN(*node*) **where**
DataCond(*node*) = **not** *waitingForStart*(*node*)
select$_{Consume}$(*node*) = $\{in \in inArc(node) \setminus atStartEnabledArc(node) \mid$
Enabled(*in*) **or** *UpstreamToken*(*in*) $\neq \emptyset\}$ // NB. all to be enabled to fire
select$_{Produce}$(*node*) = $\{o \in outArc(node) \mid DataCond(a) = true\}$
CTLOP(*node*, *I*, *O*) =
CTLOP$_{\text{GATEBEHAVIORPATTERN}}$(*node*, *I*, *O*)
waitingForStart := *true*
NormalCase(*node*) = *true* // no exception thrown in mode *reset*

7.2 Activity Behavior

ACTIVITYENTRY(*node*, *InstSet*, *TriggerProc*) = FLOWNODEBEHAVIOR(*node*)
where
CtlCond(*node*) = **forsome** $in \in inArc(node)$ *Enabled*(*in*)
CTLOP(*node*) =
let *arc* = *select*$_{Consume}$($\{in \in inArc(node) \mid Enabled(in)\}$)
CONSUME(*firingToken*(*arc*), *arc*)
let *a* = **new** *InstSet*
lifeCycle(*a*) := *ready*

```
      parent(a) := TriggerProc
    step GetActive(a, node)
GetActive(a, node) =
  if Ready(a) and forsome i ∈ inputSets(node) Available(i) then
    let i = select_InputSets({i ∈ inputSets(node) | Available(i)})
      StartExec(a, node)
      lifeCycle(a) := active
      currInputSet(node) := i
  if Interrupted(a) then Interrupt(a)
Ready(a) = (lifeCycle(a) = ready)


ActivityBehavior(node) =
  ActivityEntry(node, Instance(node, procInst(node)), procInst(node))

TaskBehavior(node) = ActivityBehavior(node) where
StartExec(a, node) = Exec(a) seq
  if Completed(a) then Exit(a, node)
  if Interrupted(a) then Interrupt(a)
  if CompensationOccurs(a) then
    TriggerCompensation(a)
    lifeCycle(a) := compensating
Exit(a, node) =
  forall o ∈ outArc(node) Produce(o)
  Delete(a, Instance(node, procInst(node)))
  PushOutput(a, node)
PushOutput(a, node) =
  if forall o ∈ outputSets(node) not Available(o)
    then Throw(noAvailOutputExc, node)
    else let o = select_OutputSets({o ∈ outputSets(node) | Available(o)})
      if IORules(node)(o, currInputSet(a)) = false
        then Throw(noIORulesExc, node)
        else Push(output(o))

Exec(t, i) = let i = currInputSet(a) in
  { Send(payload(mssg(t)), receiver(mssg(t)))   if type(t) = Send
  { Receive(mssg(t))                            if type(t) = Receive
  { Invoke(job(t), i)                           if type(t) ∈ {Service, Script}
  { Assign(job(t), i, performer(job(t), i))     if type(t) ∈ {User, Manual}
  { Call(businessRule(t), i)                    if type(t) = BusinessRule
  { skip                                        if Abstract(t)

SubProcessBehavior(node) = ActivityBehavior(node) where
StartExec(a, node) =
  if startEvent(node) ≠ undef then
    let {t} = trigger(startEvent(a))
    TriggerOccurs_P(t, startEvent(a)) := true
```

else
forall $n \in StartNode(node)$ PRODUCE$(startToken(a,node), in(n))$

EVENTSUBPROCESSBEHAVIOR$(node)$ = FLOWNODEBEHAVIOR$(node)$
where
$EventCond(node) =$
forsome $e \in StartEvent(node)$ $Happened(e)$
and if $triggeredByEvent(node)$ **then** $Active(parent(procInst(node)))$
let $e = select_{StartEvent}(\{n \in StartEvent(node) \mid Happened(e)\}$
let $\{t\} = select_{Trigger}\{t \in trigger(e) \mid TriggerOccurs(t,e)\}$
EVENTOP$(node)$ = CONSUME(t,e)
CTLOP$(node)$ =
let $P =$ **new** $Instance(process(node))$
$caller(P) := \begin{cases} parent(procInst(node)) & \textbf{if } triggeredByEvent(node) \\ caller(node) & \textbf{else} \end{cases}$
$TriggerOccurs_P(t,e) := true$
if $isInterrupting(node)$ **then** CANCEL$(parent(procInst(node)))$
$Happened(e) =$ **forsome** $t \in trigger(e)$ $TriggerOccurs(t,e)$

ADHOCBEHAVIOR$(node)$ = ACTIVITYBEHAVIOR$(node)$ **where**
STARTEXEC$(a,node)$ =
while not $AdHocCompletionCond(node)$
if $adHocOrdering(node) = Sequential$ **then** LAUNCHINNERACT$(node)$
if $adHocOrdering(node) = Parallel$ **then**
if $ActivationTime(node)$ **then** LAUNCHINNERACT$(node)$
seq
if $CancelRemainingInstances(node)$ **then**
forall $a \in RunningInnerAct(node)$
CANCEL(a)
EXIT$(a,node)$
else await forall $a \in RunningInnerAct(node)$
$Completed(a)$ **or** $Terminated(a)$
EXIT$(node)$
LAUNCHINNERACT$(node)$ =
if $enabledInnerAct(node) \neq \emptyset$ **then**
let $e = select_{EnabledInnerAct(node)}(EnabledInnerAct(node))$
ACTIVITYBEHAVIOR(e)
INSERT$(e, RunningInnerAct(node))$
DELETE$(e, EnabledInnerAct(node))$

CALLACTIVITYBEHAVIOR$(node)$ =
ACTIVITYENTRY$(node, Instance(activity(reference(node))), node)$
where STARTEXEC$(a,node)$ =
choose $n \in \{n \in StartEvent(a) \mid trigger(n) = None\}$
$TriggerOccurs_a(None,n) := true$
INSERT$(a, ActiveProcInst(activity(reference(node))))$

```
STANDARDLOOPBEHAVIOR(node) = ACTIVITYBEHAVIOR(node) where
  STARTEXEC(a,node) =
    let i = inputs(currInputSet(node))
      if testBefore(node) = true then
        while loopCond(a,node) ACTIVATE(body(a,node),i)
      if testBefore(node) = false then
        until loopCond(node) ACTIVATE(body(a,node),i)
    seq if Completed(a,node) then EXIT(a,node)
  Completed(a,node) =
    { not loopCond(a,node)  if testBefore(node) = true
    { loopCond(a,node)      if testBefore(node) = false

MULTINSTLOOPBEHAVIOR = ACTIVITYBEHAVIOR(node) where
STARTEXEC(a,node) =
  while MiCompletionCond(a,node) = false
    if isSequential(node) then
      LAUNCHINSTANCE(node) // run first instance until completion
      step  // creation of further instances
        while loopCardinality(node) >| ActiveInnerAct(a,node) |
          LAUNCHINSTANCE(a,node) // run next instance until completion
    else  // parallel case: new instances created at activation time
      while loopCardinality(node) >| ActiveInnerAct(a,node) |
        if ActivationTime(node) then  // run one more instance
          LAUNCHINSTANCE(a,node)
  step
    forall a ∈ ActiveInnerAct(a,node) CANCEL(a)
    EXIT(a,node) // NB with refined PUSHOUTPUT
LAUNCHINSTANCE(a,n) =
  let act = new Instance(innerAct(n))
    { ACTIVATE(act)
    { INSERT(act,ActiveInnerAct(a,n))
  step await Completed(act) EMITEVENT(n)
EMITEVENT(n) =
  { THROW(noneBehaviorEventRef(n),n)  if behavior(n) = None
  { THROW(oneBehaviorEventRef(n),n)   if behavior(n) = One
                                        and | Instance(innerAct(n)) |= 1
  { forall e ∈ ComplexBehaviorDefinition
      THROW(e,n)                      if behavior(n) = Complex
```

7.3 Event Behavior

```
STARTEVENTBEHAVIOR(node) = FLOWNODEBEHAVIOR(node)
  where  // normal case without parallel multiple trigger
```

EventCond(*node*) = *ParallelMultiple* ∉ *trigger*(*node*) **and**
forsome *e* ∈ *trigger*(*node*) *TriggerOccurs*(*e*, *node*)
EVENTOP(*node*) =
choose *e* ∈ {*e* ∈ *trigger*(*node*) | *TriggerOccurs*(*e*, *node*)}
CONSUME(*triggerOccurence*(*e*))
CTLOP(*node*) =
let *P* = **new** *Instance*(*process*(*node*))
forall *o* ∈ *outArc*$_P$(*node*$_P$) PRODUCE(*startToken*$_P$(*node*, *o*), *o*)

STARTEVENTPARMULTBEHAVIOR(*node*) = FLOWNODEBEHAVIOR(*node*)
where // case with parallel multiple triggers
EventCond(*node*) = *ParallelMultiple* ∈ *trigger*(*node*) **and**
forall *e* ∈ *trigger*(*node*) \ {*ParallelMultiple*}
TriggerOccurs(*e*, *node*)
EVENTOP(*node*) =
forall *e* ∈ *trigger*(*node*) \ {*ParallelMultiple*}
CONSUME(*triggerOccurence*(*e*))
CTLOP(*node*) =
let *P* = **new** *Instance*(*process*(*node*))
forall *o* ∈ *outArc*$_P$(*node*$_P$) PRODUCE(*startToken*$_P$(*node*, *o*), *o*)

ENDEVENTBEHAVIOR(*node*) = FLOWNODEBEHAVIOR(*node*) **where**
CtlCond(*node*) = **forsome** *in* ∈ *inArc*(*node*) *Enabled*(*in*)
CTLOP(*node*) = **choose** *in* ∈ {*in* ∈ *inArc*(*node*) | *Enabled*(*in*)}
CONSUME(*firingToken*(*in*), *in*)
if *Multiple* ∉ *resultType*(*node*) // normal case without multiple results
then let {*res*} = *resultType*(*node*) **in** EMITRESULT(*res*, *node*)
else forall *res* ∈ *resultType*(*node*) \ {*Multiple*}
EMITRESULT(*res*, *node*)

INTERMEDIATETHROWEVENTBEHAVIOR(*node*) =
FLOWNODEBEHAVIOR(*node*) **where**
CtlCond(*node*) = **forsome** *in* ∈ *inArc*(*node*) *Enabled*(*in*)
CTLOP(*node*) = **choose** *in* ∈ {*in* ∈ *inArc*(*node*) | *Enabled*(*in*)}
CONSUME(*firingToken*(*in*), *in*)
PRODUCEALL(*outArc*(*node*))
EVENTOP(*node*) =
if *Multiple* ∉ *trigger*(*node*) // case with only one trigger
then let {*t*} = *trigger*(*node*) **in** SETEVENTTRIGGER(*t*, *node*)
else forall *t* ∈ *trigger*(*node*) \ {*Multiple*} SETEVENTTRIGGER(*t*, *node*)
SETEVENTTRIGGER(*t*, *n*) =

$$\begin{cases} \textbf{forall } m \in \mathit{MessageFlow} \textbf{ with } \mathit{sender}(m) = \mathit{node} & \\ \quad \textsc{Send}(\mathit{payload}(m), \mathit{receiver}(m)) & \textbf{if } t = \mathit{Message} \\ \textsc{Broadcast}(\mathit{signalRef}(n), \mathit{receivers}(\mathit{signalRef}(n), n)) & \textbf{if } t = \mathit{Signal} \\ \mathit{Triggered}(\mathit{targetLink}(n)) := \mathit{true} & \textbf{if } t = \mathit{Link} \\ \textsc{Throw}(\mathit{escalation}, n) & \textbf{if } t = \mathit{Escalation} \\ \textsc{Throw}((\mathit{compensation}, \mathit{actRef}(\mathit{node})), \mathit{node}) & \textbf{if } t = \mathit{Compensation} \end{cases}$$

$\textsc{IntermediateCatchEventBehavior}(\mathit{node}) =$
  $\textsc{FlowNodeBehavior}(\mathit{node})$ **where**
$\mathit{CtlCond}(\mathit{node}) =$ **forsome** $\mathit{in} \in \mathit{inArc}(\mathit{node})$ $\mathit{Enabled}(\mathit{in})$
$\mathit{EventCond}(\mathit{node}) =$
    ($\mathit{ParallelMultiple} \notin \mathit{trigger}(\mathit{node})$ // only one trigger required to occur
    and forsome $t \in \mathit{trigger}(\mathit{node})$ $\mathit{TriggerOccurs}(t, \mathit{node})$)
  or
    ($\mathit{ParallelMultiple} \in \mathit{trigger}(\mathit{node})$ // all triggers required to occur
      and forall $t \in \mathit{trigger}(\mathit{node})$ $\mathit{TriggerOccurs}(t, \mathit{node})$)
$\textsc{EventOp}(\mathit{node}) =$
  let $\mathit{TriggOcc} = \{t \in \mathit{trigger}(\mathit{node}) \mid \mathit{TriggerOccurs}(t, \mathit{node})\}$
    if $\mathit{ParallelMultiple} \notin \mathit{trigger}(\mathit{node})$ **then**
      choose $t \in \mathit{TriggOcc}$ $\textsc{Consume}(\mathit{triggerOccurence}(t))$
    else forall $t \in \mathit{TriggOcc}$ $\textsc{Consume}(\mathit{triggerOccurence}(t))$
$\textsc{CtlOp}(\mathit{node}) =$ **choose** $\mathit{in} \in \{\mathit{in} \in \mathit{inArc}(\mathit{node}) \mid \mathit{Enabled}(\mathit{in})\}$
  $\textsc{Consume}(\mathit{firingToken}(\mathit{in}), \mathit{in})$
  $\textsc{ProduceAll}(\mathit{outArc}(\mathit{node}))$
$\mathit{TriggerOccurs}(t, \mathit{node}) =$

$$\begin{cases} \textbf{forsome } m \in \mathit{Message}\ \mathit{Received}(m, \mathit{node}) & \textbf{if } t = \mathit{Message} \\ \mathit{TimerCondition}(\mathit{node}) = \mathit{true} & \textbf{if } t = \mathit{Timer} \\ \mathit{EventExpression}(\mathit{node}) = \mathit{true} & \textbf{if } t = \mathit{Conditional} \\ \mathit{Triggered}(\mathit{sourceLink}(\mathit{node})) = \mathit{true} & \textbf{if } t = \mathit{Link} \end{cases}$$

$\textsc{BoundaryEventBehavior}(\mathit{node}) = \textsc{FlowNodeBehavior}(\mathit{node})$ **where**
$\mathit{EventCond}(\mathit{node}) =$
    ($\mathit{ParallelMultiple} \notin \mathit{trigger}(\mathit{node})$ // only one trigger required to occur
    and forsome $t \in \mathit{trigger}(\mathit{node})$ $\mathit{TriggerOccurs}(t, \mathit{node})$)
  or
    ($\mathit{ParallelMultiple} \in \mathit{trigger}(\mathit{node})$ // all triggers required to occur
      and forall $t \in \mathit{trigger}(\mathit{node})$ $\mathit{TriggerOccurs}(t, \mathit{node})$)
$\textsc{EventOp}(\mathit{node}) =$
  let $\mathit{TriggOcc} = \{t \in \mathit{trigger}(\mathit{node}) \mid \mathit{TriggerOccurs}(t, \mathit{node})\}$
    if $\mathit{ParallelMultiple} \notin \mathit{trigger}(\mathit{node})$ **then choose** $t \in \mathit{TriggOcc}$
      $\textsc{Consume}(\mathit{triggerOccurence}(t))$
      if $t = \mathit{Compensate}$ **then**
        $\textsc{Activate}(\mathit{compensation}(\mathit{attachedTo}(\mathit{node})))$
    else forall $t \in \mathit{TriggOcc}$
      $\textsc{Consume}(\mathit{triggerOccurence}(t))$

if $Compensate \in TriggOcc$ **then**
ACTIVATE($compensation(attachedTo(node))$)
CTLOP($node$) =
PRODUCEALL($outArc(node)$)
if $cancelActivity(attachedTo(node))$ **then** INTERRUPT($attachedTo(node)$)

$$TriggerOccurs(t,node) = \begin{cases} \textbf{forsome } m \in Message\ Received(m,node) & \textbf{if } t = Message \\ TimerCondition(node) = true & \textbf{if } t = Timer \\ EventExpression(node) = true & \textbf{if } t = Conditional \\ \textbf{forsome } n\ node \in receivers(signalRef(n),n) & \\ \quad \textbf{and } Arrived(signalRef(n),node) & \textbf{if } t = Signal \\ triggerOccurence(t) = (Completed,a) \textbf{ and } Completed(a) & \textbf{if } t = Compensate \\ Caught(t,node) & \\ \quad \textbf{if } t \in \{Escalation, Error, Cancel\} & \end{cases}$$

References

[Altenhofen and Börger(2009)] Altenhofen M, Börger E (2009) Concurrent abstract state machines and ^{+}CAL programs. In: Corradini A, Montanari U (eds) WADT 2008, Springer, LNCS, vol 5486, pp 1–17

[Börger and Craig(2009)] Börger E, Craig I (2009) Modeling an operating system kernel. In: Diekert V, Weicker K, Weicker N (eds) Informatik als Dialog zwischen Theorie und Anwendung, Vieweg+Teubner, Wiesbaden, pp 199–216

[Börger and Stärk(2003)] Börger E, Stärk RF (2003) Abstract State Machines. A Method for High-Level System Design and Analysis. Springer

[Börger and Thalheim(2008)] Börger E, Thalheim B (2008) A method for verifiable and validatable business process modeling. In: Börger E, Cisternino A (eds) Advances in Software Engineering, LNCS, vol 5316, Springer-Verlag, pp 59–115

[OmgBpmn(2006)] OmgBpmn (2006) Business Process Modeling Notation Specification v.1.0. dtc/2006-02-01 at http://www.omg.org/technology/documents/spec_catalog.htm

[OmgBpmn(2009)] OmgBpmn (2009) Business Process Modeling Notation (BPMN). FTF beta 1 for version 2.0. http://www.omg.org/spec/BPMN/2.0, dtc/2009-08-14

[Voelzer(2010a)] Voelzer H (2010a) A new semantics for the inclusive converging gateway in safe processes. Manuscript (submitted)

[Voelzer(2010b)] Voelzer H (2010b) Personal communication

[Wei(2010)] Wei W (2010) A translation from BPMN to Event-B. Manuscript