A Method for Verifiable and Validatable Business Process Modeling

Egon Börger[1] and Bernhard Thalheim[2]

[1] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy boerger@di.unipi.it
[2] Chair for Information Systems Engineering, Department of Computer Science, University of Kiel
D-24098 Kiel, thalheim@is.informatik.uni-kiel.de

Abstract. We define an extensible semantical framework for business process modeling notations. Since our definition starts from scratch, it helps to faithfully link the understanding of business processes by analysts and operators, on the process design and management side, by IT technologists and programmers, on the implementation side, and by users, on the application side. We illustrate the framework by a high-level operational definition of the semantics of the BPMN standard of OMG. The definition combines the visual appeal of the graph-based BPMN with the expressive power and simplicity of rule-based modeling and can be applied as well to other business process modeling notations, e.g. UML 2.0 activity diagrams.[1]

1 Introduction

Various standardization efforts have been undertaken to reduce the fragmentation of business process modeling notations and tools, most notably BPMN [15], UML 2.0 activity diagrams [1] and BPEL [2]. The main focus has been on rigorously describing the syntactical and graphical elements, as they are used by business analysts and operators to define and control the business activities (operations on data) and their (event or process driven and possibly resource dependent) execution order. Less attention has been paid to an accurate semantical foundation of the underlying concepts, which captures the interplay between data, event and control features as well as the delicate aspects of distributed computing of cooperating resource sensitive processes. We define in this paper a simple framework to describe *in application domain terms* the precise *execution semantics* of business process notations, i.e. the behavior of the described processes.

In the rest of the introduction we describe the specific methodological goals we pursue with this framework, motivate the chosen case study (BPMN) and justify the adopted method (Abstract State Machines method).

Methodological Goals. We start *from scratch*, avoiding every extraneous (read: non business process specific) technicality of the underlying computational paradigm, to faithfully capture the understanding of business processes in such a way that it can be shared by the three parties involved and serve as a solid basis for the communication between them: business analysts and operators, who work on the business process design and management side, information technology specialists, who are responsible for a faithful implementation of the designed processes, and users (suppliers and customers). From the business process management perspective it is of utmost importance to reach a transparent, easily maintainable business process documentation based upon such a common understanding (see the investigation reported in [22]).

To make the framework easily extensible and to pave the way for modular and possibly changing workflow specifications, we adopt a *feature-based* approach, where the meaning of workflow concepts

[1] The work of the first author is supported by a Research Award from the Alexander von Humboldt Foundation (*Humboldt Forschungspreis*), hosted by the Chair for Information Systems Engineering of the second author at the Computer Science Department of the University of Kiel/Germany.

can be defined elementwise, construct by construct. For each investigated control flow construct we provide a dedicated set of rules, which abstractly describe the operational interpretation of the construct.

To cope with the distributed and heterogeneous nature of the large variety of cooperating business processes, it is crucial that the framework supports descriptions that are compatible with various strategies to implement the described processes on different platforms for parallel and distributed computing. This requires the underlying model of computation to support both *true concurrency* (most general scheduling schemes) and *heterogeneous state* (most general data structures covering the different application domain elements). For this reason we formulate our descriptions in such a way that they achieve two goals:

- *separate behavior from scheduling* issues,
- *describe behavior directly in business process terms*, avoiding any form of encoding. The reason is that the adopted framework must not force the modeler to consider elements which result only from the chosen description language and are unrelated to the application problem.

Since most business process models are based on flowcharting techniques, we model business processes as diagrams (read: graphs) at whose nodes activities are executed and whose arcs are used to contain and pass the control information, that is information on execution order.[2] Thus the piecemeal definition of the behavior of single workflow constructs can be realized by nodewise defined interpreter rules, which are naturally separated from the description of the underlying scheduling scheme. Scheduling together with the underlying control flow determines when a particular node and rule (or an agent responsible for applying the rule) will be chosen for an execution step.

Case Study. As a challenging case study we apply the framework to provide a transparent accurate high-level definition of the execution semantics of the current BPMN standard, covering each of its constructs so that we obtain a complete *abstract interpreter for BPMN diagrams* (see Appendix 9). Although the BPMN standard document deals with the semantics of the BPMN elements by defining "how the graphical elements will interact with each other, including conditional interactions based on attributes that create behavioral variations of the elements" [15, p.2], this part of the specification leaves numerous questions open. For example, most attributes do not become visible in the graphical representation, although their values definitely influence the behavioral meaning of what is graphically displayed. The rules we define for each BPMN construct make all the attributes explicit which contribute to determining the semantics of the construct. This needs a framework with a sufficiently rich notion of state to make the needed attribute data available.[3]

Due to its natural-language character the BPMN standard document is also not free of a certain number of ambiguities. We identify such issues and show how they can be handled in the model we build. A summary of these issues is listed in Sect. 8.1.

For each BPMN construct we describe its behavioral meaning at a *high level of abstraction* and *piecemeal*, by dedicated transition rules. This facilitates a quick and easy reuse of the specifications when the standard definitions are completed (to fill in missing stipulations) or changed or extended. We suggest to put this aspect to use to easen the work on the planned extension of BPMN to BPMN 2.0 and to adapt the descriptions to definitions in other standards. For example, most of the rules defined in this paper or some simple variations thereof also capture the meaning of the corresponding concepts in UML 2.0 (see [38] for a concrete comparison based upon the workflow patterns in [37]). We forsee that our platform and machine independent framework can be adopted to realize the hope expressed in [37, p.25] : "Since the Activity Diagram and Business Process Diagram are very similar and are views for the same metamodel, it is possible that they will converge in the future".

A revised version BPMN 1.1 [16] of BPMN 1.0 [15] has been published after the bulk of this work had been done. The changes are minor and do not affect the framework we develop in this

[2] This does not prevent the use of dedicated arcs to represent also the data flow and other associations.
[3] The lack of state representation in BPMN is identified also in [28] as a deficit of the notation.

paper. They imply different instantiations of some of the abstractions in our BPMN 1.0 model. We therefore stick here to a model for [15].

Rational for the Method. We use the Abstract State Machine (ASM) method [12] because it directly supports the description goals outlined above: to provide for the BPMN standard a succinct, abstract and operational, easily extendable semantical model for the business process practitioner, a model he can understand directly and use to reason about his design and to hand it over to a software engineer as a binding and clear specification for a reliable and justifiably correct implementation. For the sake of easy understandability we paraphrase the formal ASM rules by verbal explanations, adopting Knuth's literate programming [26] idea to the development of specifications. Asynchronous (also called distributed) ASMs combine most general state (capturing heterogeneous data structures) with true concurrency, thus avoiding the well-known problems of Petri nets when it comes to describe complex state or non-local behavior in a distributed context (see in particular the detailed analysis in [17,36] of the problems with mapping BPMN diagrams respectively the analogous UML 2.0 activity diagrams to Petri nets).

One of the practical advantages of the ASM method derives from the fact that (asynchronous) ASMs can be operationally understood as natural extension of (locally synchronous and globally asynchronous [27]) Finite State Machines (namely FSMs working over abstract data). Therefore the workflow practitioner, supported by the common graphical design tools for FSM-like notations, can understand and use ASMs correctly as (concurrent) pseudo-code whenever there is need of an exact reference model for discussing semantically relevant issues. There is no need for any special training, besides the professional experience in process-oriented thinking. For the sake of completeness we nevertheless sketch the basic ASM concepts and our notation in an appendix, see Sect. 10.

Since ASM descriptions support an intuitive operational understanding at both high and lower levels of abstraction, the software developer can use them to introduce in a rigorously documentable and checkable way the crucial design decisions when implementing the abstract ASM models. Technically this is achieved using the ASM refinement concept defined in [8]. One can exploit this to explain how general BPMN concepts are (intended to be) implemented, e.g. at the BPEL or even lower level. In this way the ASM method allows one to add semantical precision to the comparison and evaluation of the capabilities of different tools, as undertaken in terms of natural language descriptions for a set of workflow patterns proposed for this purpose in [37,33].

The ASM method allows one to view interaction partners as rule executing agents (read: threads executing specific activities), which are subject to a separately specifiable cooperation discipline in distributed (asynchronous) runs. This supports a rigorous analysis of scheduling and concurrency mechanisms, also in connection with concerns about resources and workload balancing, issues which are crucial for (the implementation of) business processes. In this paper we will deal with multi-agent aspects only were process interaction plays a role for the behavior of a BPMN process. This is essentially via communication (messages between pools and events) or shared data, which can be represented in the ASM framework by monitored or shared locations. Therefore due to the limited support of interaction patterns in the current BPMN standard,[4] the descriptions in this paper will be mainly in terms of one process instance at a time, how it reacts to messages and events determined by and to input coming from the environment (read: other participants, also called agents). The ASM framework supports more general interaction schemes (see for example [4]).

Structure of the paper. In Sect. 2 we define the pattern for describing the semantics of workflow constructs and instantiate it in Sect. 4- 6 to define the semantics of BPMN gateways, events and activities, using some auxiliary concepts explained in Sect. 3. Appendix 9 summarizes the resulting BPMN interpreter. We discuss directly related work in Sect. 7 and suggest in Sect. 8 some further applications of our framework. Appendix 10 gives some information on the ASM method we use throughout.

[4] The BPMN standard document speaks of "different points of view" of one process by its participants, whose interactions "are defined as a sequence of activities that represent the message exchange patterns betweeen the entities involved" [15, p.11].

Our target reader is either knowledgeable about BPMN and wants to dig into (some of) its semantical intricacies, or is somebody who with the standard document at his hand tries to get a firm grasp of the semantical content of the standard definitions. This is not an introduction for a beginner.

2 The Scheme for Workflow Interpreter Rules

Data and control, the two basic computational elements, are both present in current business process models, although in the so-called workflow perspective the focus is on control (read: execution order) structures. In numerous business process notations this focus results in leaving the underlying data or resource features either completely undefined or only partly or loosely specified, so that the need is felt, when dealing with real-life business process workflows, to speak besides control patterns [37,33] separately also about data [31] and resource [32] patterns (see also [40]). The notion of abstract state coming with ASMs supports to not simply neglect data or resources when speaking about control, but to tailor their specification to the needed degree of detail, hiding what is considered as irrelevant at the intended level of abstraction but showing explicitly what is needed. For example a product catalogue is typically shared by numerous applications; it is used and manipulated by various processes, which may even be spread within one company over different and possibly also geographically separated organizational units with different access rights. In such a scenario not only the underlying data, but also their distribution and accessability within a given structure may be of importance and in need to be addressed explicitly by a business process description. A similar remark applies to the consideration of resources in a business process description. However, since in many business process notations and in particular in BPMN the consideration of resources plays no or only a minor role, we mostly disregard them here, although the framework we develop allows one to include such features.

Therefore the attention in this paper is largely reduced to control features. Business process control can be both internal and external, as usual in modern computing. The most common forms of internal, typically process-defined control encountered in workflows are sequencing, iteration, subprocess management and exception handling. They are dealt with explicitly in almost all business process notations, including BPMN, so that we will describe them in Sect. 3 as instances of the general workflow rule scheme defined below. In doing this we let the control stand out explicitly but abstractly, separating it from any form of partly data-related control.[5]

External control comes through input, e.g. messages, timers, trigger signals or conditions. This is about so-called *monitored locations*[6], i.e. variables or more generally elements of memory which are only read (not written) by the receiving agent and whose values are determined by the environment, which can be viewed as another agent. In business process notations, external control is typically dealt with by speaking of events, which we therefore incorporate into the workflow scheme below, together with resource, data and internal control features.

To directly support the widely used flowcharting techniques in dealing with business process models, we abstractly represent any business process as a graph of nodes connected by arcs, in the mathematical sense of the term. The *nodes* represent the workflow objects, where activities are performed depending on resources being available, data or control conditions to be true and events to happen, as described by transition rules associated to nodes. These rules define the meaning of workflow constructs. The *arcs* support to define the graph traversal, i.e. the order in which the workflow objects are visited for the execution of the associated rules.

For the description we use without further mentioning the usual graph-theoretic concepts, for example $source(arc)$, $target(arc)$ for source and target node of an arc, $pred(node)$ for the (possibly

[5] Counting the number of enabling tokens or considering tokens of different types in coloured Petri nets are examples of such mixed concepts of control; they are instantiations of the abstract scheme we formulate below.

[6] Concerning external control, most of what we say about monitored locations also holds for the *shared locations*, whose values can be determined by both its agent and an environment. See the ASM terminology explained in Sect. 10.

ordered) set of source nodes of arcs that have the given *node* as target node, *inArc*(*node*) for the set of arcs with *node* as target node, similarly *succ*(*node*) for the (possibly ordered) set of target nodes of arcs that have the given *node* as source node, *outArc*(*node*) for the set of arcs with *node* as source node, etc.

In general, in a given state more than one rule could be executable, even at one node. We call a node *Enabled* in a state (not to be confused with the omonymous *Enabled*ness predicate for arcs) if at least one of its associated rules is *Fireable* at this node in this state. In many applications the fireability of a rule by an agent also depends on the (degree of) availability of the needed resources, an aspect that is included into the scheme we formulate below.

The abstract scheduling mechanism to choose at each moment an enabled node and at the chosen node a fireable transition can be expressed by two here not furthermore specified selection functions, say $select_{Node}$ and $select_{WorkflowTransition}$ defined over the sets *Node* of nodes and *WorkflowTransition* of business process transition rules. These functions, whose use is supported by the notion of ASM (see Sect. 10), determine how to choose an enabled node respectively a fireable workflow transition at such a node for its execution.

WorkflowTransitionInterpreter =
let $node = select_{Node}(\{n \mid n \in Node$ **and** $Enabled(n)\})$
let $rule = select_{WorkflowTransition}(\{r \mid r \in WorkflowTransition$ **and** $Fireable(r, node)\})$
rule

Thus for every workflow construct associated to a *node*, its behavioral meaning is expressed by a guarded transition rule WorkflowTransition(*node*) $\in$ *WorkflowTransition* of the general form defined below. Every such rule states 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 ('where to proceed'),
- possibly trigger new events (besides consuming the triggering ones),
- operate on the resource space to handle (take possession of or release) resources.

In the scheme, the events and conditions in question remain abstract, the same as the operations that are performed. They can be instantiated by further detailing the guards (expressions) respectively the submachines for the description of concrete workflow transitions.[7]

WorkflowTransition(*node*) =
if *EventCond*(*node*) **and** *CtlCond*(*node*)
and *DataCond*(*node*) **and** *ResourceCond*(*node*) **then**
DataOp(*node*)
CtlOp(*node*)
EventOp(*node*)
ResourceOp(*node*)

WorkflowTransition(*node*) represents an abstract state machine, in fact a scheme (sometimes also called a pattern) for a set of concrete machines that can be obtained by further specifying the guards and the submachines. In the next section we illustrate such an instantiation process to define a high-level BPMN interpreter. For explicit instantiations of the workflow patterns in [37,33] from a few ASM workflow patterns see [10].

[7] We remind the reader that by the synchronous parallelism of single-agent ASMs, in each step all applicable rules are executed simultaneously, starting from the same state to produce together the next state.

3 Framework for BPMN Execution Model

In this section we instantiate WorkflowTransitionInterpreter to a schema for an execution model for BPMN diagrams. It is based upon the standard for the Business Process Modeling Notation (BPMN) as defined in [15]. In some cases we first formulate a general understanding of the concept in question and then explain how it can be adapted to the specific use as defined in BPMN. This is not to replace the BPMN standard, but only to provide a companion to it that explains the intended execution semantics in a rigorous high-level way and points out where attention has to be paid to the possibility of different interpretations of the standard document, due to ambiguities or underspecification. We mention here only those parts of the standard document that directly refer to the semantic behavioral interpretation of the constructs under investigation. In particular, besides what is explained in Sect. 3.1 we use numerous elements of the metamodel without further explanations, refering for their definition to the standard document.

3.1 Business Process Diagram Elements

We summarize here some of the elements which are common to every business process diagram: flow objects of various types residing at nodes connected by arcs, tokens used to represent control flow, a best practice normal form for such diagrams, etc. In a full formalization one would have to present these elements as part of a BPMN metamodel.

The graph interpretation *graph*(*process*) of a BPMN business process diagram specifies the nodes of this diagram as standing for three types of so-called flow objects, namely activities, events and gateways. We represent them as elements of three disjoint sets:

$$Node = Activity \cup Event \cup Gateway$$

To define the behavioral meaning of each BPMN flow object one may find in a *node*, we instantiate in the WorkflowTransition(*node*) scheme the guard expressions and the submachines to capture the verbal explanations produced in the standard document for each of the three flow object types. Each object type needs a specific instantiation type on can roughly describe as follows.

- To interpret the elements of the set *Event* we have to instantiate in particular the event conditions in the guard and the event operations in the body of WorkflowTransition(*node*). The instantiation of *EventCond*(*node*) interprets the cause ('trigger') of an event happening at the *node*; the instantiation of EventOp(*node*) interpretes the result ('impact') of the events (on producing other events and consuming the given ones) at this *node*.
- The interpretation of the elements of the set *Gateway* involves instantiating the guard expressions *CtlCond*(*node*) and the submachines CtlOp(*node*) of WorkflowTransition(*node*). Accompanying instantiations of *DataCond*(*node*) and DataOp(*node*) reflect what is needed in cases where also state information is involved to determine how the gateway controls the convergence or divergence of the otherwise sequential control flow.
- The interpretation of the elements of *Activity* involves instantiating the guard expressions *DataCond*(*node*) and the submachines DataOp(*node*) of WorkflowTransition(*node*). For so-called non-atomic activities, which involve subprocesses and possibly iterations over them, we will see a simultaneous instantiation also of the *CtlCond*(*node*) guards and of the submachines CtlOp(*node*) to determine the next activity.

Thus an instance of WorkflowTransitionInterpreter for BPMN diagrams is defined by instantiating a) the particular underlying scheduling mechanism (i.e. the functions $select_{Node}$ and $select_{WorkflowTransition}$) and b) WorkflowTransition(*node*) for each type of *node*. The result of such an instantiation yields a BPMN interpreter pattern, which can be instantiated to an interpreter for a particular business process diagram by further instantiating the WorkflowTransition(*node*) scheme for each concrete *node* of the diagram. This implies instantiations of the diagram related abstractions used by WorkflowTransition(*node*), as for example various attribute values. We deal with such items below as location instances, the way it is known from the object-oriented programming paradigm.

Arcs The arcs as classified into three groups, standing for the sequence flow (control flow), the message flow (data flow through monitored locations) and the associations.

The *sequence flow* arcs, indicating the order in which activities are performed in a process, will be interpreted by instantiating *CtlCond*(*node*) in the guard and CtlOp(*node*) in the body of BPMN instances of rules of form WorkflowTransition(*node*).

The *message flow* arcs define the senders and receivers of messages. In the ASM framework incoming messages represent the content of dedicated monitored locations. Sender and receiver are called participants in BPMN, in the ASM framework *agents* with message writing respectively reading rights.

Arcs representing *associations* are used for various purposes which in this paper can be mostly disregarded (except for their use for compensation discussed below)[8].

In the following, unless otherwise stated, by arc we always mean a sequence flow arc and use *Arc* to denote the set of these arcs. Many nodes in a BPMN diagram have only (at most) one incoming and (at most) one outgoing arc (see the BPMN best practice normal form below). In such cases, if from the context the *node* in question is clear, we write *in* resp. *out* instead of $inArc(node) = \{in\}$ resp. $outArc(node) = \{out\}$.

3.2 Token-Based Sequence Flow Interpretation

We mathematically represent the token-based BPMN interpretation of control flow [15, p.35] (sequence flow in BPMN terminology) by associating tokens—elements of a set *Token*—to arcs, using a dynamic function $token(arc)$.[9] 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) so that one can distinguish tokens belonging to different instances of one process *p*. Thus we write $token_{pi}$ to represent the current token marking in the process diagram instance of the process instance *pi* a token belongs to, so that $token_{pi}(arc)$ denotes the multiset of tokens belonging to process instance *pi* and currently residing on *arc*. Usually we suppress the parameter *pi*, assuming that it is clear from the context.[10]

$$token : Arc \to Multiset(Token)$$

In the token based approach to control, for a *rule* at a target node of incoming arcs to become fireable some (maybe all) arcs must be enabled by tokens being available at the arcs. This condition

[8] Association arcs in BPMN may associate semantically irrelevant additional textual or graphical information on "non-Flow Objects" to flow objects, for example so-called artifacts that provide non-functional information and "do not directly affect the execution of the Process" [15, Sect.11.12 p.182]. Association arcs may also associate processes such as compensation handlers. A typical example is a compensation intermediate event that "does not have an outgoing Sequence Flow, but instead has an outgoing directed Association" (ibid. p.133) to the target compensation activity, which is considered as being "outside the Normal Flow of the Process" (ibid. p.124). Therefore its execution effect can be disregarded for describing the semantics of BPMN—except the "flow from an Intermediate Event attached to the boundary of an activity, until the flow merges back into the Normal Flow" (ibid. p.182), which will be discussed below. Association arcs may also represent the data flow among processes, namely when they are used to describe conditions or operations on data that are involved in the activity or control flow described by the underlying flow object, as for example input/output associated to an activity (see Sect. 6). In the ASM framework these arcs point to *monitored* resp. *derived* locations, i.e. locations whose value is only read but not written resp. defined by a given scheme (see Sect. 10).

[9] We deliberately avoid introducing yet another category of graph items, like the so-called places in Petri nets, whose only role would be to hold these tokens.

[10] This treatment is in accordance with the fact that in many applications only one type of unit control token is considered, as for example in standard Petri nets. In a previous version of this paper we considered the possibility to parameterize tokens by an additional *Type* parameter, like the colours introduced for tokens in coloured Petri nets. However, this leads to add a data structure role to tokens whose intended BPMN use is to describe only "how Sequence Flow proceeds within a Process" [15, p.35].

is usually required to be an atomic quantity formula stating that the number of tokens belonging to one process instance pi and currently associated to in (read: the cardinality of $token_{pi}(in)$, denoted $\mid token_{pi}(in) \mid$) is at least the quantity $inQty(in)$ required for incoming tokens at this arc.[11] A different relation could be required, which would come up to a different specification of the predicate *Enabled*.

$Enabled(in) = (\mid token_{pi}(in) \mid \geq inQty(in)$ **forsome** $pi)$

Correspondingly the control operation CtlOp of a workflow usually consists of two parts, one describing how many tokens are Consumed on which incoming arcs and one describing how many tokens are Produced on which outgoing arcs, indicated by using an analogous abstract function *outQty*. We use macros to encapsulate the details. They are defined first for consuming resp. producing tokens on a given arc and then generalized for producing or consuming tokens on a given set of arcs.

Consume$(t, in) =$ Delete$(t, inQty(in), token(in))$
Produce$(t, out) =$ Insert$(t, outQty(out), token(out))$
Pass$(t, in, out) =$
  Consume(t, in)
  Produce(t, out)

In various places the BPMN standard document alludes to structural relations between the consumed incoming and the produced outgoing tokens. To express this we use 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, all belonging to the same process instance pi and ready to be Consumed. For the sake of exposition we make the usual assumption that $inQty(in) = 1$, so that we can use the following sequence notation:

$$firingToken([a_1, \ldots, a_n]) = [t_1, \ldots, t_n]$$

to denote that t_i is the token selected to be fired on arc a_i. We write $firingToken(in) = t$ instead of $firingToken(\{in\}) = [t]$.

If one considers, as seems to be very often the case, only (multiple occurrences of) indistinguishable tokens, all belonging to one process instance, instead of mentioning the single tokens one can simplify the notation by parameterizing the macros only by the arcs:

Consume$(in) =$ Delete$(inQty(in), token(in))$
Produce$(out) =$ Insert$(outQty(out), token(out))$
ConsumeAll$(X) =$ **forall** $x \in X$ Consume(x)
ProduceAll$(Y) =$ **forall** $y \in Y$ Produce(y)

Remark. This use of macros allows one to adapt the abstract token model to different instantiations by a concrete token model. For example, if a token is defined by two attributes, namely the process instance pi it belongs to and the arc where it is *pos*itioned, as seems to be popular in implementations, then it suffices to refine the macro for Passing a token t from in to out by updating the second token component, namely from its current *pos*ition value in to its new value out:

Pass$(t, in, out) = (pos(t) := out)$

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. Note that 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 their cumulative effect is considered.

[11] The function *inQty* generalizes the *startQuantity* attribute for activities in the BPMN standard.

Four Instantiation Levels Summarizing the preceding discussion one sees that the structure of our model provides four levels of abstraction to separate different concerns, among them the distinction between process and process instances.

- At the first level, in WorkflowTransitionInterpreter, scheduling is separated (via functions $select_{Node}$ and $select_{WorkflowTransition}$) from behavior (via rules in *WorkflowTransition*).
- At the second level, different constructs are behaviorally separated from each other by defining a machine pattern for each construct type—here gateways, events and activities—instantiating appropriately the components of the abstract machine WorkflowTransition(*node*) as intended for each type.
- At the third level, a concrete business process is defined by instantiating the per *node* globally defined rule pattern WorkflowTransition(*node*) for each concrete diagram node.
- At the fourth level, instances of a concrete business process are defined by instantiating the attributes and the *token* function as instance locations belonging to the process instance. In object-oriented programming terms one can explain the last two steps as adding to static class locations (the global process attributes) dynamic instance locations (the attribute instantiations).

BPMN Token Model The BPMN standard document uses a more elaborate concept of tokens, though it claims to do this only "to facilitate the discussion" of "how Sequence Flow proceeds within a Process". The main idea is expressed as follows:

> The behavior of the Process can be described by tracking the path(s) of the Token through the Process. A Token will have a unique identity, called a TokenId set, that can be used to distinguish multiple Tokens that may exist because of concurrent Process instances or the dividing of the Token for parallel processing within a single Process instance. The parallel dividing of a Token creates a lower level of the TokenId set. The set of all levels of TokenId will identify a Token. [15, p.35]

The standard document imposes no further conditions on how to realize this token traceability at gateways, activities, etc., but uses it for example for the tracing of structured elements in the mapping from BPMN to BPEL (op.cit.pg.192 sqq.). For the sake of completeness we illustrate here one simple structure-based formalization of the idea of tokens as a hierarchy of sets at different levels, which enables the designer to convey non-local information between gateways.[12]

The goal is to directly reflect the use of tokens for tracing the sequence flow through starting, splitting, joining, calling (or returning from), iterating, ending processes, instantiating multiple instances of an activity or otherwise relating different computation paths. At the first level one has (a finite multiset of occurrences of) say one basic token $origin(p)$, containing among other data the information on the process ID of the process p upon whose start the token has been created. These tokens are simply passed at all nodes with only one incoming and one outgoing arc (see the remark on tokens at intermediate events at the end of Sect. 5). When it comes to "the dividing of the Token for parallel processing within a single Process instance", the considered (multiset of occurrences of the) token t is Consumed and Produces the (multiset of the desired number of occurrences of) next-level tokens $par(t, p(i), m)$, one for each of the m parallel processes $p(i)$ in question for $0 < i < m$. When (the considered number of occurrences of) these tokens arrive on the arcs leading to the associated (if any) join node, (the multisets of) their occurrences are Consumed and the (desired number of occurrences of the) *parent token* t is (re-) Produced.

In the same manner one can also distinguish tokens $andSplitToken(t, i, m)$ for AND-split gateways, $orSplitToken(t, i, m)$ for OR-split gateways, $multInstToken(t, i)$ or $multInstToken(t, i, m)$ for multi-instances of a (sub)process, etc. One can also parameterize the tokens by the nodes where they are produced or let m be a dynamic function of the parameters of the considered diagram node (gateway instance). Using a tree structure for the representation of such token functions allows the

[12] For another possibility one can use in dynamic contexts, where there is no possibility to refer to static structural net information, see the remark in Sect. 4 on relating OR-split and OR-joins.

workflow designer to define in a simple, intuitive and precise way any desired notion of "parallel" paths. It also supports a computationally inexpensive definition of a process to be *Completed* when all its tokens have been consumed, since the relation between a token and the process ID of the process p where it has been created is given by the notion of $origin(p)$ in the token set $T(p)$.

3.3 BPMN Best Practice Normal Form

For purely expository purposes, but without loss of generality, we assume BPMN graphs to be in (or to have been equivalently transformed into) the following normal form, in [15] called 'modeling covenience':

- *BPMN Best Practice Normal Form.* [15, p.69] Disregarding arcs leading to exception and compensation nodes, only gateways have multiple incoming or multiple outgoing arcs. Except so-called complex gateways, gateways never have both multiple incoming and multiple outgoing arcs.

Justification. We outline the proof idea for some characteristic cases; the remaining cases will be considered at the places where the normal form is used to shorten the descriptions. An AND (also called conjunctive or parallel) gateway with n incoming and m outgoing arcs can be transformed into a standard equivalent graph consisting of a parallel AND-Join gateway with n incoming and one outgoing arc, which is incoming arc to a parallel AND-Split gateway with m outgoing arcs. A so-called uncontrolled node with n incoming and m outgoing arcs can be shown to be standard equivalent to an OR-Join gateway with n incoming arcs connected by one outgoing arc to a new node which is connected to an AND-Split gateway with m outgoing arcs. If one is interested in a completely carried out formal description of the behavior of all BPMN constructs, one has to add to the behavioral descriptions we give in this paper a description of the transformation of arbitrary BPMN diagrams into diagrams in BPMN Best Practice Normal Form. This is a simple exercise.

4 BPMN Execution Model for Gateway Nodes

Gateways are used to describe the convergence (merging) or divergence (splitting) of control flow in the sense that tokens can 'be merged together on input and/or split apart on output' [15, p.68]. Both merging and splitting come in BPMN in two forms, which are considered to be related to the propositional operators **and** and **or**, namely

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

For the conjunctive case the BPMN terminology is 'forking' ('dividing of a path into two or more parallel paths, also known as an AND Split') [15, p.110] respectively 'parallel joining' (AND-Join). For the disjunctive case the BPMN standard distinguishes two forms of split, depending on whether the decision among the alternatives is exclusive (called XOR-Split) or not (called OR-Split, this case is also called 'inclusive'). For the exclusive case a further distinction is made depending on whether the decision is 'data-based' or 'event-based'. These distinctions are captured in the instantiations of WorkflowTransition($node$) for gateway $node$s below by corresponding $EventCond(node)$ and $DataCond(node)$ guards, which represent these further gateway fireability conditions, besides the mere sequence flow enabledness.

The BPMN standard views gateways as 'a collection of *Gates*' that are associated one-to-one to outgoing sequence flow arcs of the gateway, 'one Gate for each outgoing Sequence Flow of the Gateway' [15, p.68]. The sequence flow arcs are required to come with an expression that describes the condition under which the corresponding gate can be taken.[13] Since this distinction is not needed for a description of the gateway behavior, we abstract from it in our model and represent

[13] The merge behavior of an OR gateway is represented by having multiple incoming sequence flow, as formalized by *CtlCond* below, but only one gate (with its associated sequence flow condition set to None, realizing that the condition is always true).

gates simply by the outgoing sequence flow arcs to which they are associated. Nevertheless, for the sake of a clear exposition of the different split/merge features, we start from the BPMN best practice normal form assumption whereby each gateway performs only one of the two possible functions, either divergence or convergence of multiple sequence flow. For the special case of gateways without incoming arcs or without outgoing arcs, which play the role of start or end events, see the remarks at the end of the section on start and end events. The gateway pattern definition we present in Sect. 4.6 for the so-called complex gates (combinations of simple decision/merge) makes no normal form assumption, so that its scheme shows ho to describe gateways that are not in normal form. From a definition of the complex case one can easily derive a definition of the simple cases, as we will see below.

4.1 AND-Split (Fork) Gateway Nodes

By the normal form assumption, every AND-split gateway *node* has one incoming arc *in* and finitely many outgoing arcs. Therefore *CtlCond*(*node*) is simply *Enabled*(*in*). CtlOp(*node*) means to Consume(t, *in*) for some enabling token t chosen from *token*(*in*) and to Produce on each outgoing arc o the (required number of) *andSplitToken*(t, o) (belonging to the same process instance as t), which in the case of unit tokens are simply occurences of t.

In BPMN DataOp(*node*) captures multiple assignments that may be 'performed when the Gate is selected' [15, Table 9.30 p.86] (read: when the associated rule is fired). We denote these assignments by sets *assignments*(o) associated to the outgoing arcs o (read: gates).

Thus the WorkflowTransition(*node*) scheme is instantiated for any **and**-split (fork) gateway *node* as follows:

AndSplitGateTransition(*node*) = WorkflowTransition(*node*)
where
CtlCond(*node*) = *Enabled*(*in*)
CtlOp(*node*) =
let t = *firingToken*(*in*)
Consume(t, *in*)
ProduceAll({(*andSplitToken*(t, o), o) | $o \in$ *outArc*(*node*)})
DataOp(*node*) = //performed for each selected gate
forall $o \in$ *outArc*(*node*) **forall** $i \in$ *assignments*(o) Assign($to_i, from_i$)

This is still a scheme, since for each particular diagram node for example the source and target expressions $to_i, from_i$ for the associated assignments have still to be instantiated.

4.2 AND-Join (Synchronization) Gateway Nodes

By the normal form assumption, every AND-join gateway *node* has finitely many incoming and one outgoing arc. Each incoming arc is required to be *Enabled*, so that *CtlCond*(*node*) is simply the conjunction of these enabledness conditions. CtlOp(*node*) means to Consume firing tokens (in the requested quantity) from all incoming arcs and to Produce (the considered number of) *andJoinToken*s on the outgoing arc, whose values depend on the incoming tokens. DataOp(*node*) captures multiple assignments as in the case of AND-split gateways.[14]

Remark. If AND-join nodes n' are structural companions of preceding AND-split nodes n, the tokens t_j = *andSplitToken*(t, o_j) produced at the outgoing arc o_j of n will be consumed at the corresponding arc in_j incoming n', so that at the arc outgoing n' the original token t will be produced. Such a structured relation between splits and joins is however not prescribed by the BPMN standard, so that for the standard the functions *andSplitToken* and *andJoinToken* remain abstract (read: not furthermore specified, i.e. freely interpretable by every standard conform implementation).

[14] If our understanding of the BPMN standard document is correct, the standard does not forsee event-based or data-based versions for AND-join transitions, so that the conditions *EventCond*(*node*) and*DataCond*(*node*) and the EventOp can be skipped (or set to true resp. **skip** for AND-joins).

AndJoinGateTransition($node$) = WorkflowTransition($node$)
where
$CtlCond(node)$ = **forall** $in \in inArc(node)$ $Enabled(in)$
CtlOp($node$) =
let $[in_1, \ldots, in_n] = inArc(node)$
let $[t_1, \ldots, t_n] = firingToken(inArc(node))$
ConsumeAll($\{(t_j, in_j)) \mid 1 \leq j \leq n\}$)
Produce($andJoinToken(\{t_1, \ldots, t_n\}), out$)
DataOp($node$) = **forall** $i \in assignments(out)$ Assign($to_i, from_i$)

4.3 OR-Split Gateway Nodes

An OR-split node is structurally similar to an AND-split node in the sense that by the normal form assumption it has one incoming and finitely many outgoing arcs, but semantically it is different since instead of producing tokens on every outgoing arc, this may happen only on a subset of them.

The chosen alternative depends on certain conditions $OrSplitCond(o)$ to be satisfied that are associated to outgoing arcs o. In BPMN the choice among these alternatives is based either upon process-data-involving *GateCond*itions that evaluate to true (data-based case) or upon *GateEvent*s that are *Triggered* (event-based case). Further variants considered in BPMN depend upon whether at each moment exactly one alternative is chosen (the exclusive case) or whether more than one of the alternative paths can be taken (so-called inclusive case).

We formulate the choice among the alternatives by an abstract function $select_{Produce}(node)$, which is constrained to select at each invocation a non-empty subset of arcs outgoing *node* that satisfy the *OrSplitCond*ition. If there is no such set, the rule cannot be fired.

Constraints for $select_{Produce}$
$select_{Produce}(node) \neq \emptyset$
$select_{Produce}(node) \subseteq \{out \in outArc(node) \mid OrSplitCond(out)\}$[15]

This leads to the following instantiation of the WorkflowTransition($node$) scheme for **or**-split gateway *node*s. The involvement of process data or gate events for the decision upon the alternatives is formalized by letting *DataCond* and *EventCond* in the rule guard and their related operations in the rule body depend on the parameter O for the chosen set of alternatives. As done for AND-split nodes, we use an abstract function *orSplitToken* to describe the tokens Produced on the outgoing arc; in general their values depend on the incoming tokens.

OrSplitGateTransition($node$) =
let $O = select_{Produce}(node)$ **in** WorkflowTransition($node, O$)
where
$CtlCond(node)$ = $Enabled(in)$
CtlOp($node, O$) =
let $t = firingToken(in)$
Consume(t, in)
ProduceAll($\{(orSplitToken(t, o), o) \mid o \in O\}$)
DataOp($node, O$) = **forall** $o \in O$ **forall** $i \in assignments(o)$ Assign($to_i, from_i$)

From OrSplitGateTransition also AndSplitGateTransition can be defined by requiring the selection function to select the full set of all outgoing arcs.

[15] Instead of requiring this constraint once and for all for each such selection function, one could include the condition as part of $DataCond(node, O)$ resp. $EventCond(node, O)$ in the guard of OrSplitGateTransition.

4.4 OR-Join Gateway Nodes

As for AND-join gateway nodes, by the normal form assumption, every OR-join gateway *node* has finitely many incoming and one outgoing arc. Before proceeding to deal with the different cases the BPMN standard names explicitly (exclusive and data-based or event-based inclusive OR), we formulate a general scheme from which the BPMN instances can be derived.

For OR-join nodes one has to specify what happens if the enabledness condition is satisfied simultaneously for more than one incoming arc. Should all the enabling tokens from all enabled incoming arcs be consumed? Or only tokens from one enabled arc? Or from some but maybe not all of them? Furthermore, where should the decision about this be made, locally by the transition rule or globally by the scheduler which chooses the combination? Or should assumptions on the runs be made so that undesired combinations are excluded (or proved to be impossible for a specific business process)? More importantly one also has to clarify whether firing should wait for other incoming arcs to get enabled and in case for which ones.

To express the choice of incoming arcs where tokens are consumed we use an abstract selection function $select_{Consume}$: it is required to select a non-empty set of enabled incoming arcs, whose enabling tokens are consumed in one transition, if there are some enabled incoming arcs; otherwise it is considered to yield the empty set for the given argument (so that the rule which is governed by the selection via the $CtlCond(node)$ is not fireable). In this way we explicitly separate the two distinct features considered in the literature for OR-joins: the enabledness condition for each selected arc and the synchronization condition that the selected arcs are exactly the ones to synchronize. The convential token constraints are represented as part of the control condition in the OrJoinGateTransition rule below, namely that the selected arcs are all enabled and that there is at least one enabled arc. What is disputed in the literature and not specified in the BPMN standard is the synchronization constraint for $select_{Consume}$ functions. Therefore we formulate the transition rule for an abstract OR-join semantics, which leaves the various synchronization options open as additional constraints to be put on $select_{Consume}$. As a result $select_{Consume}(node)$ plays the role of an interface for triggering for a set of to-be-synchronized incoming arcs the execution of the rule at the given *node*.

This leads to the following instantiation of the WorkflowTransition($node$) scheme for **or**-join gateway *node*s. To abstractly describe the tokens Produced on the outgoing arc we use a function *orJoinToken* whose values depend on the tokens on the selected incoming arcs.

OrJoinGateTransition($node$) =
  let $I = select_{Consume}(node)$ **in** WorkflowTransition($node, I$)
where
  $CtlCond(node, I) =$ ($I \neq \emptyset$ **and forall** $j \in I$ $Enabled(j)$)
  CtlOp($node, I$) =
    Produce($orJoinToken(firingToken(I)), out$)
    ConsumeAll($\{(t_j, in_j) \mid 1 \leq j \leq n\}$) **where**
      $[t_1, \ldots, t_n] = firingToken(I)$
      $[in_1, \ldots, in_n] = I$
  DataOp($node$) = **forall** $i \in assignments(out)$ Assign($to_i, from_i$)

NB. Clearly AndJoinGateTransition, in BPMN called the merge use of an AND-gateway, can be defined as a special case of the merge use of an OR-gateway OrJoinGateTransition, namely by requiring the selection function to always yield either the empty set or the set of all incoming arcs.

Remark on relating OR-Split and OR-Joins The discussion of "problems with OR-joins" has received much attention in the literature, in particular in connection with EPCs (Event driven Process Chains) and Petri nets (see for example [25,42] and the references there). In fact, to know how to define the choice function $select_{Consume}$ is a critical issue also for (implementations of) the BPMN standard. The BPMN standard document seems to foresee that the function is dynamic so

that it does not depend only on the (static) diagram structure. In fact the following is required: "Process flow SHALL continue when the signals (Tokens) arrive from all of the incoming Sequence Flow that are expecting a signal based on the upstream structure of the Process ... Some of the incoming Sequence Flow will not have signals and the pattern of which Sequence Flow will have signals may change for different instantiations of the Process." [15, p.80] Generally it is claimed in the literature that the "non-locality leads to serious problems when the formal semantics of the OR-join has to be defined" [24, p.3]. The discussion of and the importance attached to these "problems" in the literature is partly influenced by a weakness of the underlying Petri-net computation model, which has well-known problems when dealing with non-local (in particular if dynamic) properties.[16] In reality the issue is more a question of process design style, where modeling and verification of desired process behavior go hand in hand via modular (componentwise) definitions and reasoning schemes, which are not in need of imposing static structural conditions (see [5]). It is only to a minor extent a question of *defining* the semantics of OR-joins. In fact, in the ASM framework one can succinctly describe various ways to dynamically relate the tokens produced by an OR-Split node to the ones consumed by an associated OR-Join node. See [14].

4.5 BPMN Instances of Gateway Rules

In BPMN gateways are classified into exclusive (XOR), inclusive (OR), parallel (AND) and complex. The case of complex gateways is treated below.

An AND gateway of BPMN can be used in two ways. When it is used 'to create parallel flow', it has the behavior of AndSplitGateTransition, where each outgoing arc represents a gate (without loss of generality we assume the BPMN Best Practice Normal Form, i.e. a gateway with one incoming arc). The so-called merge use of the AND gateway of BPMN 'to synchronize parallel flow' has the behavior of AndJoinGateTransition.

The data-based XOR and the OR gateway of BPMN, when 'acting only as a Merge', both have only one gate without an associated *GateCond* or *GateEvent*; the event-based XOR is forbidden by the standard to act only as a Merge. Thus those two gateway uses are an instance of OrJoinGateTransition where $select_{Consume}$ is restricted to yield either an empty set (in which case the rule cannot fire) or

- for XOR a singleton set,
- for OR a subset of the incoming arcs with associated tokens 'that have been produced upstream' [15, p.80][17].

This satisfies the standard document requirement for XOR that in case of multiple incoming flow, the incoming flow which in a step of the gateway has not be chosen 'will be used to continue the flow of the Process (as if there were no Gateway)'; similarly for OR [15, p.75].

When acting as a split into alternatives, the XOR (in its two versions data-based and event-based) and the OR gateway of BPMN are both an instance of OrSplitGateTransition where $select_{Produce}$ is restricted to yield one of the following:

- For the data-based XOR a singleton set consisting of the first $out \in outArc(node)$, in the given order of gates, satisfying $GateCond(out)$. If our understanding of BPMN is correct then in this case $DataCond(node, O) = GateCond(out)$ and $EventCond(node, O) = true$.
- For the event-based XOR a singleton set O consisting of the first $out \in outArc(node)$, in the given order of gates, satisfying $GateEvent(out)$. If our understanding of BPMN is correct then in this case $EventCond(node, O) = GateEvent(out)$ and $DataCond(node, O) = true$.
- For OR a non-empty subset of the outgoing arcs.

[16] Similar problems have been identified in [36] for the mapping of UML 2.0 activity diagrams to Petri nets.

[17] The BPMN document provides no indications for determining this subset, which has a synchronization role.

4.6 Gateway Pattern (Complex Gateway Nodes)

Instead of defining the preceding cases separately one after the other, one could define once and for all *one* general gateway pattern that covers the above cases as well as what in BPMN are called complex gateway nodes, namely by appropriate configurations of the pattern abstractions. This essentially comes up to define two general machines CONSUME and PRODUCE determining the (possibly multiple) incoming respectively outgoing arcs where tokens are consumed and produced. The abstract function *patternToken* determines which tokens are produced on each outgoing arc in relation to the *firingToken*s on the incoming arcs I. As shown above it can be refined for specific gateway nodes, for example for OR-split/join gateways to *orSplit*/*JoinToken*, for AND-split/join gateways to *andSplit*/*JoinToken*, etc.

GATETRANSITIONPATTERN(*node*) =
  let $I = select_{Consume}(node)$
  let $O = select_{Produce}(node)$ **in**
    WORKFLOWTRANSITION(*node*, I, O)
where
  $CtlCond(node, I) =$ $(I \neq \emptyset$ **and forall** $in \in I$ $Enabled(in))$
  CTLOP(*node*, I, O) =
    PRODUCEALL($\{(patternToken(firingToken(I), o), o) \mid o \in O\}$)
    CONSUMEALL($\{(t_j, in_j) \mid 1 \leq j \leq n\}$) **where**
      $[t_1, \ldots, t_n] = firingToken(I)$
      $[in_1, \ldots, in_n] = I$
  DATAOP(*node*, O) = **forall** $o \in O$ **forall** $i \in assignments(o)$ ASSIGN($to_i, from_i$)

From this GATETRANSITIONPATTERN(*node*) machine one can define the machines above for the various simple gateway nodes. For AND-joins $select_{Consume}$ chooses all incoming arcs, whereas for OR-joins it chooses exactly (exclusive case) or at least one (inclusive cases). Similarly $select_{Produce}$ chooses all the outgoing arcs for AND-split gateways and exactly one (exclusive case) or at least one outgoing arc (inclusive case) for OR-split nodes, whether data-based or event-based.

Remark. As mentioned already above, the BPMN standard document allows gateway nodes to be without incoming or without outgoing arc. To such nodes the general stipulations on BPMN constructs without incoming or without outgoing arc in relation to start or end events apply, which are captured in our model as described in the two remarks at the end of the sections on start and end events below.

5 BPMN Execution Model for Event Nodes

Events in BPMN can be of three types, namely *Start*, *Intermediate* and *End* events, intended to "affect the sequencing or timing of activities of a process" [15, Sect.9.3]. Thus BPMN events correspond to internal states of Finite State Machines (or more generally control states of control-state ASMs [7], see Sect. 10), which start/end such machines and manage their intermediate control states. So the set *Event* is a disjoint union of three subsets we are going to describe now.

$Event = StartEvent \cup IntermEvent \cup EndEvent$

5.1 Start Events

A start event has no incoming arc ('no Sequence flow can connect to a Start Event').[18] Its role is to indicate 'where a particular Process will start'. Therefore a start event, when *Triggered*— a monitored predicate representing that the event " "happens" during the course of a business process" [15, Sect.9.3]— generates a token (more generally: the required quantity of tokens) on an outgoing arc. This is expressed by the transition rule STARTEVENTTRANSITION(*node*, e) defined

[18] For one exception to this discipline see below.

below, an instance of WorkflowTransition(*node*) where data and control conditions and data operations are set to empty since they are unrelated to how start events are defined in BPMN.

By *trigger*(*node*) we indicate the set of types of (possibly multiple) event *trigger*s that may be associated to *node*, each single one of which can be one of the following: a message, a timer, a condition (in the BPMN document termed a rule), a link or none. The BPMN standard document leaves it open how to choose a single one out of a multiple event associated to a node in case two or more events are triggered there simultaneously. This means that the non-deterministic choice behavior is not furthermore constrained, so that we use the ASM **choose** operator to select a single event trigger and thereby a rule StartEventTransition(*node*, *e*) for execution, each of which is parameterized by a particular event $e \in trigger(node)$.[19] This reflects the standard requirement that "Each Start Event is an independent event. That is, a Process Instance SHALL be generated when the Start Event is triggered." [15, p.36][20]

StartEventTransition(*node*) =
  choose $e \in trigger(node)$ StartEventTransition(*node*, *e*)

By the best practice normal form we can assume that there is exactly one outgoing arc *out*, namely after replacing possibly multiple outgoing arcs by one outging arc, which enters an and-split gateway with multiple outgoing arcs. This captures that by the BPMN standard document "Multiple Sequence Flow MAY originate from a Start Event. For each Sequence Flow that has the Start Event as a source, a new parallel path SHALL be generated ... Each path will have a separate unique Token that will traverse the Sequence Flow." [15, Sect.9.3.2 p.38-39] Therefore a StartEventTransition(*node*, *e*) rule fires when the *EventCond*(*node*) is true that *e* is *Triggered*. It yields as event EventOp(*node*, *e*) to ConsumEvent(*e*) and

StartEventTransition(*node*, *e*) rule yields as CtlOp(*node*) to Produce a *startToken* on *out*. The produced token is supposed to contain the information needed for "tracking the path(s) of the Token through the Process" [15, p.35]. Since this information is not furthermore specified by the standard document, in our model it is kept abstract in terms of an abstract function *startToken*(*e*). Traditionally it is supposed to contain at least an identifier for the just startede process instance.

StartEventTransition(*node*, *e*) =
  if *Triggered*(*e*) **then** Produce(*startToken*(*e*), *out*)
                ConsumEvent(*e*)

Remark to event consumption in the start rule. If the intention of the standard document is that not only the chosen triggered event but all triggered events are consumed, it suffices to replace ConsumEvent(*e*) by the following rule:

forall $e' \in trigger(node)$ **if** *Triggered*(e') **then** ConsumEvent(e')

.

The definition of *Triggered*(*e*) is given by Table 9.4 in [15].

The submachine ConsumEvent(*e*) is defined depending on the type of event *e*. Messages and timers represent (values of) monitored locations with a predetermined consumption procedure. The standard document leaves it open whether upon firing a transition triggered by an incoming message, that message is consumed or not.[21] Similarly it is not specified whether a timer event is automatically consumed once its time has passed (precisely or with some delay). Therefore for the BPMN 1.0 standard, for these two cases the submachine ConsumEvent remains abstract, it has to be specified by the intended consumption discipline of each system instance.

The same holds for events of type None or Rule.

[19] An alternative would be to use a (possibly local and dynamic) selection function $select_{Event}$ which each time chooses an event out of the set *trigger*(*node*).

[20] See also the remark below.

[21] This is an important issue to clarify, since a same message may be incoming to different events in a diagram.

Events e of type Link are used "for connecting the end (Result) of one Process to the start (Trigger) of another" [15, Sect.9.3.2 pg.37]. In accordance with the interpretation of a Link Intermediate Event as so-called "Off-Page connector" or "Go To" object [15, Sect.9.3.4 p.48] we represent such links as special sequence flow arcs, connecting $source(link)$ ("one Process") to $target(link)$ ("another Process", in the BPMN standard denoted by the attribute $ProcessRef(node)$) with $token$ defined for some $linkToken(link)$. Therefore $Triggered(e)$ for such a start event means $Enabled(link)$ and the ConsumEvent submachine deletes $linkToken(link)$, which has been produced before on this $link$ arc at the $source(link)$, as result of a corresponding end event or link event at the source link of a paired intermediate event (see below). Thus we have the following definition for start events e of type Link (we write $link$ for the connecting arc corresponding to the type Link):

if $type(e) = Link$ **then**
  $Triggered(e) = Enabled(link)$
  ConsumEvent($link$) = Consume($linkToken(link), link$)

There is one special case where a start event e can have a virtual incoming arc $inarc(e)$, namely "when a Start Event is used in an Expanded Sub-Process and is attached to the boundary of that Sub-Process". In this case "a Sequence Flow from the higher-level Process MAY connect to the Start Event in lieu of connecting to the actual boundary of the Sub-Process" [15, Sect.9.3.2 pg. 38]. This can be captured by treating such a connection as a special arc $inarc(e)$ incoming the start event e, which is enabled by the higher-level Process via appropriate $subProcTokens$ so that it suffices to include into the definition of $Triggered(e)$ for such events the condition $Enabled(inarc(e))$ and to include into ConsumEvent(e) an update to Consume($subProcToken(e), inarc(e)$).

Remark on processes without start event. There is a special case that applies to various BPMN constructs, namely items that have no incoming arc (sequence flow) and belong to a *process without start event*. They are required by the standard document to be activated (performed) when their process is instantiated. For the sake of exposition, to avoid having to deal separately for each item with this special case, we assume without loss of generality that each process has a (virtual) start event and that all the items without incoming sequence flow included in the process are connected to the start event by an arc so that their performance is triggered when the start node is triggered by the instantiation of the process. One could argue in favor of including this assumption into the BPMN Best Practice Normal Form.

Remark on multiple start events. For a later version of the standard it is contemplated that there may be "a dependence for more than one Event to happen before a Process can start" such that "a correlation mechanism will be required so that different triggered Start Events will apply to the same process instance." [15, p.36-37] For such an extension it suffices to replace in StartEventTransition the non-deterministically chosen event by a set of $CorrelatedEvents$ as follows:

MultipleStartEventTransition($node$) =
  choose $E \subseteq CorrelatedEvent(node)$
    MultipleStartEventTransition($node, E$)
MultipleStartEventTransition($node, E$) =
  if forall $e \in E$ $Triggered(e)$ **then**
    Produce($startToken(e), out$)
    forall $e \in E$ ConsumEvent(e)

Remark The instantiation mechanism of BPMN using an event-based gateway with its attribute "instantiate" set to "true" is covered by the semantics as defined here for start events.

5.2 End Events

End events have no outgoing arc ("no Sequence Flow can connect from an End Event"). "An End Event MAY have multiple incoming Sequence Flow. The Flow MAY come from either alternative or parallel paths... If parallel Sequence Flow target the End Event, then the Tokens will be consumed

as they arrive" [15, Sect.9.3.3 p.42,40]. This means that also for describing the behavior of end event nodes we can assume without loss of generality the best practice normal form, meaning here that there is exactly one incoming arc *in*—namely after replacing possibly multiple incoming arcs by one arc that is incoming from a new or-join gateway, which in turn is entered by multiple arcs (equipped with appropriate associated token type). Thus an end event transition fires if the *CtlCond* is satisfied, here if the incoming arc is *Enabled*; as CtlOp it will Consume(*in*) the firing token. BPMN forsees for end events also a possible EventOperation, namely to EmitResult of having reached this end event of the process instance to which the end event node belongs, which is assumed to be encoded into the firing token. We use a function *res*(*node*) to denote the result defined at a given node.

EndEventTransition(*node*) =
  if *Enabled*(*in*) **then**
    Consume(*firingToken*(*in*), *in*)
    EmitResult(*firingToken*(*in*), *res*(*node*), *node*)

The type of result and its effect are defined in [15, Table 9.6]. We formalize this by a submachine EmitResult. It Sends messages for results of type Message, where Send denotes an abstract message sending mechanism (which assumes the receiver information to be retrievable from the message). In case of Error, Cancel or Compensation type, via EmitResult an intermediate event is *Triggered* to catch the error, cancel the transaction or compensate a previous action. We denote this intermediate event, which is associated in the diagram to the considered *node* and the type of *result*, by *targetIntermEv*(*result*, *node*).[22] The node to which *targetIntermEv* belongs is denoted by *targetIntermEvNode*(*res*, *node*). In the Cancel case also "A Transaction Protocol Cancel message should be sent to any Entities involved in the Transaction" [15, Sect.9.3.3 Table 9.6], formalized below as a Callback to *listener*(*cancel*, *node*). Receiving such a message is presumably supposed to have as effect to trigger a corresponding intermediate cancel event (see [15, p.60]).

A result of type Link is intended to connect the end of the current process to the start of the target process. This leads us to the end event counterpart of the formalization explained above for start events of type Link: an end event node of type Link is the *source*(*link*) of the interpretation of *link* as a special sequence flow arc, where by the rule WorkflowTransition(*source*(*link*)) the *linkTokens*, needed to make the link *Enabled*, are Produced. As we will see below this may also happen at the source link of a paired intermediate event node of type Link. These tokens will then be consumed by the rule WorkflowTransition(*target*(*link*)) at *target*(*link*), e.g. a connected start event node of type Link whose incoming arc has been *Enabled*. We use the same technique to describe that, in case the result type is None and *node* is a subprocess end node, "the flow goes back to its Parent Process": we Produce appropriate tokens on the *targetArc*(*node*), which is supposed to lead back to the node where to return in the *parent*(*p*) process.

For a result of type Terminate we use a submachine DeleteAllTokens that ends all activities in the current process instance, including all multiple instances of activities, by deleting the tokens from the arcs leading to such activities. To denote these activities we use a set *Activity*(*p*) which we assume to a) contain all activities contained in process instance *p* and b) to be dynamically updated by all running instances of multiple instances within *p*. In defining DeleteAllTokens we also reflect the fact that tokens are viewed in the BPMN standard as belonging to the process in which they are created—"an End event consumes a Token that had been generated from a Start Event within the same level of Process" [15, Sect.9.3.3 p.40]. Therefore we delete not all tokens, but only all tokens belonging to the given process *p*, denoted by a set *TokenSet*(*p*).

For the Multiple result type we write *MultipleResult*(*node*) for the set of single results that are associated to the *node*: for each of them the EmitResult action is taken.

[22] In case of Error this intermediate event is supposed to be within what is called the *Event Context*, in case of Cancel it is assumed to be attached to the boundary of the Transaction Sub-Process where the Cancel event occurs.

EmitResult(t, $result$, $node$) =
  if $type(result) = Message$ **then** Send($mssg(node, t)$)
  if $type(result) \in \{Error, Cancel, Compensation\}$ **then**
    $Triggered(targetIntermEv(result, node)) := true$ // trigger intermediate event
    Insert($exc(t)$, $excType(targetIntermEvNode(result, node))$)
  if $type(result) = Cancel$ **then**
    Callback($mssg(cancel, exc(t), node)$, $listener(cancel, node)$)
  if $type(result) = Link$ **then** Produce($linkToken(result)$, $result$)
  if $type(result) = Terminate$ **then** DeleteAllTokens($process(t)$)
  if $type(result) = None$ **and** $IsSubprocessEnd(node)$ **then**
    Produce($returnToken(targetArc(node), t)$, $targetArc(node)$)
  if $type(result) = Multiple$ **then**
    forall $r \in MultipleResult(node)$ EmitResult($t, r, node$)
where
  Callback(m, L) = **forall** $l \in L$ Send(m, l)
  DeleteAllTokens(p) = **forall** $act \in Activity(p)$
    forall $a \in inArc(act)$ **forall** $t \in TokenSet(p)$ Empty($token(a, t)$)

This concludes the description of end events in BPMN, since "Flow Objects that do not have any outgoing Sequence Flow" but are not declared as end events are treated the same way as end events. In fact "a Token entering a path-ending Flow Object will be consumed when the processing performed by the object is completed (i.e., when the path has completed), as if the Token had then gone on to reach an End Event." [15, Sect.9.3.3 pg.40-41]. This is captured by the CtlOp($node$) submachine executed by the WorkflowTransition($node$) rule for the corresponding $node$ to Consume(in) when $Enabled(in)$.

Remark on tokens at start/end events. The standard document explains tokens at end events as follows:

> ... an End Event consumes a Token that had been generated from a Start Event within the same level of Process. If parallel Sequence Flow target the End Event, then the Tokens will be consumed as they arrive. [15, p.40]

Such a constraint on the tokens that are Produced at a start event to be Consumed at end events in possibly parallel paths of the same process level comes up to a specification of the abstract functions denoting the specific tokens associated to the arc outgoing start events respectively the arc incoming end events.

Remark on Process Completion. For a process to be *Completed* it is required that "all the tokens that were generated within the Process must be consumed by an End Event", except for subprocesses which "can be stopped prior to normal completion through exception Intermediate Events" (ibid.). There is also the special case of a process without end events. In this case, "when all Tokens for a given instance of the Process are consumed, then the process will reach a state of being completed" (ibid., p.41). It is also stipulated that "all Flow Objects that do not have any outgoing Sequence Flow ... mark the end of a path in the Process. However, the process MUST NOT end until all parallel paths have completed" (ibid., p.40), without providing a definition of "parallel path". This issue should be clarified in the standard document. For some of the BPMN constructs there is a precise definition of what it means to be *Completed*, see for example the case of task nodes below.

5.3 Intermediate Events

In BPMN intermediate event nodes are used in two different ways: to represent exception or compensation handling (Exception Flow Case) or to represent what is called Normal Flow (Normal Flow Case). In the first case the intermediate event e is placed on the boundary of the task or sub-process to which the exception or compensation may apply. $targetAct(e)$ denotes the activity

to whose boundary e is attached and for which it "is used to signify an exception or compensation" [15, Sect.9.3.4 Table 9.9]. We denote such events as *BoundaryEv*ents. They do not have any ingoing arc ("MUST NOT be target for Sequence Flow"), but typically have one outgoing arc denoted again by *out* ("MUST be a source for Sequence Flow; it can have one (and only one) outgoing Sequence Flow", except for intermediate events of type Compensation which "MAY have an outgoing Association") [15, Sect.9.3.4 p.47]. In the Normal Flow Case the intermediate event occurs "in the main flow" of the process (not on the boundary of its diagram) and has a) exactly one *out*going arc,[23] b) exactly one *in*going arc if it is of type None, Error or Compensation and at most one *in*going arc if it is of type Message, Timer, Rule or Link.

The behavioral meaning of an intermediate event also depends on the associated event type, called trigger [15, Sect.9.3.4 Table 9.8]. As for start events, we use *trigger*(*node*) to indicate the set of types of (possibly multiple) event *trigger*s that may be associated to *node*. For intermediate events, in addition to the types we saw for start events, there are three (trigger) types that are present also for end events, namely Error, Cancel and Compensation. Following Table 9.8 and the specification of the Activity Boundary Conditions in op.cit., intermediate events of type Error, Compensation, Rule, Message or Timer can be used in both the Normal Flow and the Exception Flow case, whereas intermediate events of type None or Link are used only for Normal Flow and intermediate events of type Cancel or Multiple only for *BoundaryEv*ents.

If two or more event triggers are *Triggered* simultaneously at an intermediate event node, since "only one of them will be required", one of them will be chosen, the same as established for start event nodes. (As we will see below, for intermediate events type Multiple is allowed to occur only on the boundary of an activity.)

IntermEventTransition(*node*) =
  choose $e \in$ *trigger*(*node*) IntermEventTransition(*node*, e)

It remains therefore to define IntermEventTransition(*node*, e) for each type of event e and depending on whether e is a *BoundaryEv*(e) or not.

In each case, the rule checks that the event is *Triggered*. The definition of *Triggered*(e) given for start events in Table 9.4 of [15] is extended in Table 9.8 for intermediate events to include the types Error, Cancel and Compensation. An intermediate event of type *Cancel* is by definition in [15, Sect.9.3.4 Table 9.8] a *BoundaryEv*ent of a transaction subprocess and *Triggered* by an end event of type *Cancel* or a Callback message received during the execution of the transaction. Similarly an intermediate event of type Error or Compensation can be *Triggered* in particular as the result of an end event of corresponding type, see the definition of EmitResult for end events. The EventOp(*node*) will ConsumEvent(e), which is defined as for start events adding for the three event types Error, Cancel and Compensation appropriate clauses (typically the update *Triggered*(e) := *false*).

In the Normal Flow Case where *BoundaryEv*(e) is false, the rule guard contains also the *CtlCond* that the *in*coming arc—where the activity was waiting for the intermediate event to happen—is *Enabled*. Correspondingly there is a CtlOp(*node*) to Consume(*in*). Where the sequence flow will continue depends on the type of event.

In case of an intermediate event of type *Link*, the considered *node* is the source link node of a paired intermediate event and as such has to Produce(*linkToken*(*link*), *link*), read: the appropriate link token(s) on the link—which is interpreted in our model as a special arc that leads to the target link node of the paired intermediate event, as explained above for start and end events.

Case *type*(e) = *None* is meant to simply "indicate some state of change in the process", so that the CtlOp will also Produce an appropriate number and type of tokens on the *out*going arc. The same happens in case of an intermediate event of type Message or Timer.

An intermediate event of type Error or Compensation or Rule within the main flow is intended to "change the Normal Flow into an Exception or Compensation Flow", so that the error or compensation is Thrown, which means that the corresponding next enclosing *BoundaryEv*ent

[23] Except source link intermediate events, which therefore receive a special treatment in rule IntermEventTransition(*node*, e) below.

occurrence (which we denote by a function *targetIntermEv* similar to the one used already in EmitResult above) is *Triggered* to handle (catch or forward) the exception, error (corresponding to the *ErrorCode* if any) or compensation. In addition the information on the token that triggered the event is stored in the *targetIntermEv* by inserting it into a set *excType*, which is used when the boundary intermediate event is triggered.

In the Exception Case where $BoundaryEv(e)$ is true, if the activity to whose boundary the intermediate event is attached is *active*,[24] the sequence flow is requested to "change the Normal Flow into an Exception Flow" and to TryToCatch the exception respectively perform the compensation. If there is no match for the exception, it is rethrown to the next enclosing corresponding intermediate *BoundaryEv*ent. If the match succeeds, the *out* arc (which we interprete in our model as an association arc in case of a compensation) leads in the diagram to an exception handling or compensation or cancelling activity and the CtlOp(*node*) action consists in making this arc *Enabled* by an operation Produce(*out*).

Every intermediate event of type Compensation attached to the boundary of an activity is assumed by BPMN to catch the compensation (read: to satisfy *ExcMatch*) since "the object of the activity that needs to be compensated ... will provide the Id necessary to match the compensation event with the event that "threw" the compensation". For transactions the following is required:

> When a Transaction is cancelled, then the activities inside the Transaction will be subjected to the cancellation actions, which could include rolling back the process and compensation for specific activities ... A Cancel Intermediate Event, attached to the boundary of the activity, will direct the flow after the Transaction has been rolled back and all compensation has been completed. [15, p.60]

The standard document does not specify the exact behavior of transactions[25] and refers for this as an open issue to an Annex D (ibid.), but this annex seems to have been removed and not be accessible any more. We therefore formulate only the cited statement and leave it as an open issue how the cancellation activities (roll back and/or compensation) are determined and their execution controlled.

```
IntermEventTransition(node, e) =
  if Triggered(e) then
    if not BoundaryEv(e) then
      if Enabled(in) then let t = firingToken(in)
        ConsumeEvent(e)
        Consume(t, in)
        if type(e) = Link then Produce(linkToken(link), link)
        if type(e) = None then Produce(t, out)
        if type(e) = Message then
          if NormalFlowCont(mssg(node), process(t))
            then Produce(t, out)
            else Throw(exc(mssg(node)), targetIntermEv(node))
        if type(e) = Timer then Produce(timerToken(t), out)
        if type(e) ∈ {Error, Compensation, Rule} then Throw(e, targetIntermEv(e))
    if BoundaryEv(e) then
```

[24] The boundary creates what is called the Event Context. "The Event Context will respond to specific Triggers to interrupt the activity and redirect the flow through the Intermediate Event. The Event Context will only respond if it is active (running) at the time of the Trigger. If the activity has completed, then the Trigger may occur with no response." [15, Sect.10.2.2 p.131]

[25] Also the descriptions in Table.8.3 (p.15), Table 8.3 (p.25) and Table B.50 (p.271, related to the attributes introduced in Table 9.13 (p.56)) are incomplete, as is the description of the group concept introduced informally in Sect.9.7.4 (p.95-97). The latter permits a transaction to span over more than one process, without clarifying the conditions for this by more than the statement that "at the end of a successful Transaction Sub-Process ... the transaction protocol must verify that all the participants have successfully completed their end of the Transaction" (p.61).

```
    if active(targetAct(e)) then
      ConsumEvent(e)
      if type(e) = Timer then Insert(timerEv(e), excType(node))
      if type(e) = Rule then Insert(ruleEv(e), excType(node))
      if type(e) = Message then Insert(mssgEv(e), excType(node))
      if type(e) = Cancel then choose exc ∈ excType(node) in
        if Completed(Cancellation(e, exc)) then Produce(excToken(e, exc), out)
      else TryToCatch(e, node)
where
  TryToCatch(ev, node) =
    if ExcMatch(ev) then Produce(out(ev))
      else TrytoCatch(ev, targetIntermEv(node, ev))
  Completed(Cancellation(e)) =
    RolledBack(targetAct(e)) and Completed(Compensation(targetAct(e)))
```

Remark. For intermediate events of type Message, Timer, Rule or Link the BPMN standard allows the event to be without incoming arc and to "always be ready to accept the Event Triggers while the Process in which they are contained is active" [15, Sect.9.3.4 p.48]. In this case we understand the IntermEventTransition($node, e$) rule as being written without Consume(t, in) and with the guard $Enabled(in)$ replaced by $active(targetAct(e))$. $ExcMatch(e)$ is assumed to be true for each *Triggered* event of type Timer, Message or Rule.

The above formalization captures that an intermediate event on the boundary of a process which contains an externally executed task can be triggered by the execution of that task. In fact the atomicity of tasks does not imply their zero-time execution.[26]

Remark on token passing. Differently from gateway nodes, where the consumed and the produced tokens may carry different information, and differently from start or end event nodes where tokens are only produced or only consumed, for intermediate event nodes a typical assumption is that tokens are simply passed. A similar remark applies to all nodes with only one incoming and one outgoing arc (see for example the activity nodes below).

6 BPMN Execution Model for Activity Nodes

Activities are divided into two types, atomic activities (*tasks*) and compound ones (*subprocesses*). Both tasks and subprocesses can contain iterative components of different *loopType*, namely so-called *standard* loops (*while*, *until*) or *multiInstance* loops (*for each*); for subprocesses this includes also so-called ad-hoc processes. For purely expository purposes, to avoid repetitions, we therefore slightly deviate from the classification in the standard document and put these iterative tasks or subprocesses into a third category of say iterative processes (IterProc), without changing any of their standard attributes. Therefore we have the following split of *Activity* into three subsets:

$$Activity = Task \cup SubProcess \cup IterProc$$
$$IterProc = Loop \cup MultiInstance \cup AdHoc$$

The notion of atomicity is the one known from information systems, meaning that the task in question "is not broken down to a finer level of Process Model detail" [15, Sect.9.4.3 p.62]; it does not imply the 0-time-execution view that is traditionally associated with the notion of atomicity. Typically the action underlying the given task is intended to represent that within the given business process "an end-user and/or an application are used to perform the Task when it is executed" (ibid.), so that atomicity refers to the fact that as part of a business process the task is viewed as a unit process and not "defined as a flow of other activities"(ibid.p.53), though it may and usually will take its execution time without this time being furthermore analyzed in the workflow

[26] The Petri net model for tasks in [17] is built upon the assumption that "the occurrence of the exception may only interrupt the normal flow at the point when it is ready to execute the task". But this seems to be an over-simplification of exceptions triggered by tasks.

diagram.[27] We reflect this notion of atomicity by using in the definition of TaskTransition(*task*) below the sequentiality operator **seq** for structuring ASMs (see [12, Ch.4]). This operator 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, exactly as required by the BPMN understanding of task execution.

Besides being "defined as a flow of other activities" to achieve modularity of the process design, compound subprocesses are also used to a) create a context for exception handling and compensation (in a transactional context) "that applies to a group of activities", b) for a compact representation of parallel activities and c) for process instantiation, as will be discussed below.

Every activity comes with finitely many (possibly zero) associated so-called InputSets and OutputSets, which define the data requirements for input to and output from the activity. When these sets are present, at least one input must be defined "to allow the activity to be performed" and "at the completion of the activity, only one of the OutputSets may be produced", the choice being up to the implementation—but respecting the so-called IORules, expressions that "may indicate a relationship between an OutputSet and an InputSet that started the activity" [15, Sect.9.4.3 Table 9.10].

6.1 Task Nodes

In this section we consider only tasks that are not marked as iterative; tasks and subprocesses marked as Loop or MultInstance are considered below.

For the sake of simplicity of exposition in the following description we assume also for tasks the BPMN Best Practice Normal Form for sequence flow connections, namely that tasks have (at most) one incoming arc and (at most) one outgoing arc. In fact, multiple incoming flow, which may be from alternative or from parallel paths,[28] can be taken care of by adding a preceding OR-Join respectively AND-Join gateway node; multiple outgoing flow can be taken care of by adding a following AND-Split gateway, so that "a separate parallel path is being created for each Flow" [15, p.67-68].

Thus in case *in*coming and/or *out*going arcs are present, the TaskTransition(*task*) rule has as *CtlCond*(*task*) the guard *Enabled*(*in*) and as CtlOp(*task*) the machines Consume(*in*) and/or Produce(*out*). By including in the definition below these control parts into square brackets we indicate that they may not be there, depending on whether the considered *task* node has *in*coming and/or *out*going arcs or not. Since the execution of the action associated to the task may take time, the action Produce(*out*) to forward the control should take place only after that execution has *Completed*(*task*), together with the (possibly missing) output producing action ProduceOutput(*outputSets*(*task*)) defined below. Therefore every rule TaskTransition(*task*) will consist of sequentially first Execuing the task proper and then, upon task completion, proceeding to produce the output (if any) and the tokens (in case) to forward the control.

Whether a rule TaskTransition(*task*) can be fired depends also on a *DataCond*(*task*) expressing that the *task* is *ReadyForExec*ution, which in turn depends on the particular type of the *task*, as does the task Execution. The standard considers eight types for tasks:

$$TaskType = \{Service, User, Receive, Send, Script, Manual, Reference, None\}$$

A *task* of type Service or User is defined to be *ReadyForExec*ution upon "the availability of any defined InputSets", formalized by a predicate *SomeAvail*(*inputSets*(*task*)) to be true. To Exec(*task*) in these two cases means to Send(*inMssg*(*task*)) ("at the start of the Task"). In the Service case this is presumably intended to have the effect to Activate the associated service, characterized as "some sort of service, which could be a Web service or an automated application" [15, p.64]; in the User case presumably to Activate the external *performer*s of the associated *action* for the given input, characterized as "the human resource that will be performing the User Task

[27] This may also explain why a BPMN task is allowed to have an iterative substructure.
[28] In the case of alternative paths the standard documents speaks of *uncontrolled flow*.

... with the assistance of a software application" (ibid.p.65-66). [29] In both cases to ACTIVATE the (performance of the) task is followed by waiting until an $outMssg(task)$ arrives that "marks the completion of the Task".[30] The latter is formalized by the predicate $Completed(task)$ [15, Table 9.18 p.64, Table 9.21 p.66].

A *task* of type Receive "is designed to wait for a message to arrive ... Once the message has been received, the Task is completed." Therefore EXEC($task$) is defined as RECEIVE($mssg(task)$) and $ReadyForExec(task)$ is defined as $Arrived(mssg(task))$. There is a special case that a Receive task is "used to start a Process", which is indicated by an attribute called $Instantiate(task)$. In this case it is required for the underlying diagram, as static constraint, that either *task* has no *in*coming arc and the associated process has no start event, or *task* has an *in*coming arc and $source(in)$ is a start event of the associated process [15, Table 9.19 p.65]. Therefore in this particular case $ReadyForExec(task)$ is defined to be the conjunction of $Instantiate(task) = true$ and $Arrived(mssg(task))$.

Tasks of type Send, Manual or Script are designed to unconditionally EXECute the associated action, namely to SEND($msgg(task)$) respectively to CALL the performer(s) of the associated manual action or script code—presumably with the effect to trigger its execution and to wait until that action or code execution is *Completed*. In the case of script code the executing agent (read: the engine that interpretes the script code) is the *performer* and the script code represents the to be executed *action*. In the case of a manual task, to CALL the *performer* is intended to activate "the human resource that will be performing the Manual Task" [15, Table 9.23 p.67], which we denote as *action* of the task for the given input.

A task of type Reference simply calls another task; to EXECute it means to EXECute the referenced $taskRef(task)$ (recursive definition).

The standard document determines the $currInput(task)$, from where the (assumed to be defined) $inputs(currInput(task))$ to start *task* are taken, by saying that "each InputSet is sufficient to allow the activity to be performed" [15, Table 9.10 p.50], leaving it open which element of $inputSets(task)$ to choose if there are more than one available. We therefore consider $currInput(task)$ as result of an implementation-defined selection procedure $select_{InputSets}$ that selects an element out of $SomeAvail(inputSets(task))$. This input remains known until the end of the proper task EXECution since the choice of the output may depend on it via the relation $IORules(task)$ between input and output sets (see below the definition of PRODUCEOUTPUT).

To produce an output (if any, indicated in the definition of TASKTRANSITION($task$) by square brackets) upon task completion,[31] an element of $outputSets(task)$ with defined output is chosen that satisfies the $IORule(task)$ together with the $currInputSet(task) \in inputSets(task)$ from which the inputs had been taken to start the *task*. For the chosen element the defined $outputs(o)$ are EMITted [15, Table 9.10 p.50].

We collect here also the BPMN stipulations for the completion of single tasks.

$$Completed(t, ttype) = \begin{cases} Arrived(outMssg(t, ttype)) & \textbf{if } type(t) \in \{Service, User\} \\ Received(mssg(task, ttype)) & \textbf{if } type(t) = Receive \\ Sent(mssg(task, ttype)) & \textbf{if } type(t) = Send \\ Completed(action(t, inputs(currInput(t))), ttype) & \textbf{if } type(t) \in \{Script, Manual\} \\ Completed(taskRef(t), ttype) & \textbf{if } type(t) = Reference \end{cases}$$

[29] The standard document leaves it open whether the service executing agent respectively the human *performer*s are incorporated as address into the $inMssg(task)$ or whether this address should be a parameter of the SEND machine.

[30] It remains unclear in the wording of the standard document whether *Arrived* or *Received* is meant here.

[31] In case of no *out*going sequence flow and no end event in the associated process, the task (if it is not marked as a Compensation Task, in which case it is "not considered a part of the Normal Flow") "marks the end of one or more paths in the Process." In this case the process is defined to be completed "when the Taks ends and there are not other parallel paths active" [15, Table 9.4.3 p.68]. This definition assumes the *other parallel paths* to be known, although from the standard document it is not clear whether this knowledge derives from static information on the graph structure or from run-time bookkeeping of the paths that form a parallel subprocess. Presumably it is intended to permit both.

Besides the notions of messages to have *Arrived* or been *Sent* they use a concept of completion for the execution of (the actions associated to) script and manual tasks, all of which the standard document seems to assume as known.

TaskTransition($task$) = [**if** *Enabled*(in) **then**]
  if *ReadyForExec*($task$) **then let** t = *firingToken*(in)
    [Consume(t, in)]
    let $i = select_{InputSets}(SomeAvail(inputSets(task)))$
      Exec($task, inputs(i)$)
      $currInput(task) := i$
  [**seq**
    if *Completed*($task, t$) **then**
      [ProduceOutput($outputSets(task), currInput(task)$)]
      [Produce($taskToken(task, t), out$)]]
where
  ProduceOutput($outputSets(t), i$) =
    choose $o \in outputSets(t)$ **with** *Defined*($outputs(o)$) **and** $IORules(t)(o, i) = true$
      Emit($outputs(o)$)

$$ReadyForExec(t) = \begin{cases} SomeAvail(inputSets(t)) & \textbf{if } type(t) \in \{Service, User\} \\ Arrived(mssg(t)) \text{ [}\textbf{and } Instantiate(t)\text{]} & \textbf{if } type(t) = Receive \\ true & \textbf{if } type(t) \in \{Send, Script, Manual, Reference\} \end{cases}$$

$$\text{Exec}(t, i) = \begin{cases} \text{Send}(inMssg(t)) & \textbf{if } type(t) \in \{Service, User\} \\ \text{Receive}(mssg(t)) & \textbf{if } type(t) = Receive \\ \text{Send}(mssg(t)) & \textbf{if } type(t) \in \{Send\} \\ \text{Call}(performer(action(t, i)), action(t, i)) & \textbf{if } type(t) \in \{Script, Manual\} \\ \text{Exec}(taskRef(t), i) & \textbf{if } type(t) = Reference \\ \textbf{skip} & \textbf{if } type(t) = None \end{cases}$$

6.2 Iterative Activity Nodes

The BPMN concepts of iterative activities correspond to well-known programming concepts of iterated, parallel or sequential execution or stepwise execution in a non-deterministic order. Nevertheless we include their discussion here for the sake of completeness. Except their internal iterative structure, iterative activities (tasks and subprocesses with corresponding markers) share the general sequence flow and input/output mechanism of arbitrary activities. Therefore we reuse in the transition rules for iterative activities the corresponding (possibly missing, depending on whether there is incoming or outgoing sequence flow) entry and exit part of the TaskTransition($task$) rule without further explanations. For the sake of exposition we assume without loss of generality also for iterative activity nodes the BPMN Best Practice Normal Form so that we consider (at most) one *in*going and (at most) one *out*going arc.

Standard Loops Each activity in the set *Loop* of standard loops comes with a *loopCond*ition that may be evaluated at one of the following two moments (called *testTime*):

- *before* the to be iterated *act*ivity begins, in which case the loop activity corresponds to the programming construct **while** *loopCond* **do** *act*,
- *after* the activity finishes, in which case the loop activity corresponds to the programming construct **until** *loopCond* **do** *act*.

The BPMN standard forsees also that in each round a *loopCounter* is updated, which can be used in the *loopCond* (as well as a *loopMaximum* location). The standard document does not explain however whether the input is taken only once, at the entry of the iteration, or at the beginning of each iteration step. There are reasonable applications for both interpretations, so that the issue should be clarified. This is partly a question of whether the function *inputs*, which is applied to the selected input set *currInput*(*node*) to provide the input for the *iterBody* of the to be iterated activity, is declared to be a static or a dynamic function.

The preceding discussion is summarized by the following rule for *node*s with *loopType*(*node*) = *Standard*. For a natural definition of **while** and **until** in a way that is compatible with the synchronous parallelism of ASM execution see [12, Ch.4]. We use an abstract function *loopToken* to denote how (if at all) the information on loop instances and incoming tokens is elaborated during the iteration.

```
LoopTransition(node) = [if Enabled(in) then]
  let t = firingToken(in)
    LoopEntry(node, t)
    seq
      if testTime(node) = before then
        while loopCond(node, t) LoopBody(node, t)
      if testTime(node) = after then
        until loopCond(node, t) LoopBody(node, t)
    [seq LoopExit(node, t)]
where
  LoopBody(n, t) =
    loopCounter(node, t) := loopCounter(node, t) + 1
    iterBody(node, loopToken(t, loopCounter(node, t) + 1)[, inputs(currInput(node))])
```

The auxiliary machines LoopEntry and LoopExit are defined as follows (the possibly missing parts, in case there is no incoming/outgoing sequence flow or no input/output, are in square brackets). Note that the predicate *LoopCompleted*(*n*) is not defined in the standard document. It seems that the standard permits to exit a loop at any place, for example by a link intermediate event (Fig.10.46 p.126) or by a so-called Go To Object (Fig.10.45 ibid.), so that the question has to be anwered whether this is considered as completion of the loop or not (see the example for "improper looping" in Fig.10.51 p.129).

```
LoopEntry(n, t) =
  loopCounter(n, t) := 0
  [Consume(t, in)]
  [currInput(n) := select_InputSets(SomeAvail(inputSets(n)))]
LoopExit(n, t) =
  if Completed(n, t) then
    [ProduceOutput(outputSets(n), currInput(n))]
    [Produce(loopExitToken(t, loopCounter(n, t)), out)]
Completed(n, t) = LoopCompleted(n, t) if n ∈ Loop(t)
```

Multi-Instance Loops The iteration condition of activities in the set *MultiInstance* of multi-instance loops is integer-valued, an expression (location in ASM terms) denoted *miNumber*, called MI-Condition in the standard document. A *miOrdering* for the execution of the instances is defined, which is either parallel or sequential. In the latter case the order seems to implicitly be understood as the order of integer numbers, so that we can use for the description of this case the ASM construct **foreach** (for a definition see the appendix Sect. 10) followed by the submachine LoopExit defined above. Also in this case a *loopCounter* is "updated at runtime", though here it is allowed to only be "used for tracking the status of a loop" and not in *miNumber*, which is assumed to be "evaluated only once before the activity is performed" [15, Sect.9.4.1]. We reflect in the rule

MultiInstTransition below the explicitly stated standard requirement that "The LoopCounter attribute MUST be incremented at the start of a loop".

In the parallel case a *miFlowCond*ition indicates one of four types to complete the parallel execution of the multiple instances of *iterBody*. In these four cases we know only that all iteration body instances are started in parallel (simultaneously). Therefore we use an abstract machine Start for starting the parallel execution of the multiple instances of the iteration body. The requirements for the *miOrdering* = *Parallel* case appear in [15, Table 9.12 p.52] and read as follows.

- Case *miFlowCond* = *All*: "the Token SHALL continue past the activity after all of the activity instances have completed". This means to LoopExit(*node*) only after for each $i \leq miNumber$ the predicate $Completed(iterBody(node, miToken(t, i)[\ldots]))$ has become true.
  Note that for the (sequential or parallel) splitting of multiple instances the information on the current multiple instance number i becomes a parameter of the *miToken* function in the iteration body; it corresponds to (and typically will be equal to) the $loopCounter(node, t)$ parameter of the *loopToken* function in LoopTransition. In this way the token $miToken(t, i)$ contains the information on the current iteration instance.
- Case *miFlowCond* = *None*, also called *uncontrolled flow*: "all activity instances SHALL generate a token that will continue when that instance is completed". This means that each time for some $i \leq miNumber$ the predicate $Completed(iterBody(node, miToken(t, i)[\ldots]))$ becomes true, one has to Produce a token on *out*. We define below a submachine EveryMultInstExit to formalize this behavior.
- Case *miFlowCond* = *One*: "the Token SHALL continue past the activity after only one of the activity instances has completed. The activity will continue its other instances, but additional Tokens MUST NOT be passed from the activity". We define below a submachine OneMultInstExit to formalize this behavior.
- Case *miFlowCond* = *Complex*: a *complexMiFlowCond* expression, whose evaluation is allowed to involve process data, "SHALL determine when and how many Tokens will continue past the activity". Thus *complexMiFlowCond* provides besides the number *tokenNo* (of activity instances that will produce continuation tokens) also a predicate *TokenTime* indicating when passing the token via Produce(*out*) is allowed to happen. We will formalize the required behavior in a submachine ComplMultInstExit defined below. There it will turn out that EveryMultInstExit and OneMultInstExit are simple instantiations of ComplMultInstExit.

```
MultiInstTransition(node) = [if Enabled(in) then]
  let t = firingToken(in)
    LoopEntry(node, t)
    seq
      if miOrdering(node) = Sequential then
        foreach i ≤ miNumber(node)
          loopCounter(node, t) := loopCounter(node, t) + 1
          iterBody(node, miToken(t, i)[, inputs(currInput(node))])
        seq LoopExit(node, t)
      if miOrdering(node) = Parallel then
        forall i ≤ miNumber(node)
          Start(iterBody(node, miToken(t, i)[, inputs(currInput(node))]))
        seq
          if miFlowCond = All then
            if Completed(node, t) then LoopExit(node, t)
          if miFlowCond = None then EveryMultInstExit(node, t)
          if miFlowCond = One then OneMultInstExit(node, t)
          if miFlowCond = Complex then ComplMultInstExit(node, t)
  where
    Completed(n, t) = forall i ≤ miNumber(n) Completed(iterBody(n, miToken(t, i)[...]))
```

ComplMultInstExit has to keep track of whether the initially empty set of those activity instances, which have *AlreadyCompleted* and have passed their continuation tokens to the outgoing arc, has reached the prescribed number *tokenNo*(*complexMiFlowCond*) of elements. If yes, the remaining instances upon their completion are prevented from passing further tokens outside the multiple instance activity. If not, each time an instance appears to be in *NewCompleted* we once more Produce a token on the outgoing arc *out*—if the *TokenTime*(*complexMiFlowCond*) condition allows us to do so, in which case we also insert the instance into the set *AlreadyCompleted*. Since the context apparently is distributed and since the standard document contains no constraint on *TokenTime*(*complexMiFlowCond*), at each moment more than one instance may show up in *NewCompleted*.[32] Therefore we use a selection function $select_{NewCompleted}$ to choose an element from the set *NewCompleted*[33] of multiple instances that have *Completed* but not yet produced their continuation token.[34] In the following definition n is supposed to be a multiple instance activity node with parallel *miOrdering*. The standard document leaves it open whether output (if any) is produced either after each instance has completed or only at the end of the entire multiple instance activity, so that in our definition we write the corresponding updates in square brackets to indicate that they may be optional.

ComplMultInstExit(n, t) = // for $miOrdering(n) = Parallel$
  $AlreadyCompleted := \emptyset$ // initially no instance is completed
  seq
    while $AlreadyCompleted \neq \{i \mid i \leq miNumber(n)\}$ **do**
      if $NewCompleted(n, t) \neq \emptyset$ **then**
        if $\mid AlreadyCompleted \mid < tokenNo(complexMiFlowCond)$
        then
          if $TokenTime(complexMiFlowCond)$ **then**
            let $i_0 = select_{NewCompleted}$ **in**
              Produce$(miExitToken(t, i_0), out)$
              Insert$(i_0, AlreadyCompleted)$
            [ProduceOutput$(outputSets(n), currInput(n))$]
          else forall $i \in NewCompleted(n, t)$ Insert$(i, AlreadyCompleted)$
where
$NewCompleted(n, t) =$
  $\{i \leq miNumber(n) \mid Completed(iterBody(n, miToken(t, i)[\ldots])) \textbf{ and } i \notin AlreadyCompleted\}$

The EveryMultInstExit machine is an instance of ComplMultInstExit where *tokenNo* is the number (read: cardinality of the set) of all to-be-considered activity instances and the *TokenTime* is any time.

EveryMultInstExit(n, t) = ComplMultInstExit(n, t)
  where
    $tokenNo(complexMiFlowCond) = \mid \{i \mid i \leq miNumber(n)\} \mid$
    $TokenTime(complexMiFlowCond) = true$

[32] The description of the case *miFlowCond* = *One* in the standard document is ambiguous: the wording *after only one of the activity instances has completed* seems to implicitly assume that at each moment at most one activity instance can complete its action. It is unclear whether this is really meant and if yes, how it can be achieved in a general distributed context.

[33] In ASM terminology this is a derived set, since its definition is fixed and given in terms of other dynamic locations, here *Completed* and *AlreadyCompleted*.

[34] If one prefers not to describe any selection mechanism here, one could instead use the **forall** construct as done in the **else** branch. This creates however the problem that it would not be impossible for more than *tokenNo*(*complexMiFlowCond*) many process instances to complete simultaneously so that a more sophisticated mechanism must be provided to limit the number of those ones that are allowed to Produce a token on the outgoing arc.

OneMultInstExit is an instance of ComplMultInstExit where $tokenNo = 1$ and the $TokenTime$ is any time.

OneMultInstExit(n, t) = ComplMultInstExit(n, t)
  where
    $tokenNo(complexMiFlowCond) = 1$
    $TokenTime(complexMiFlowCond) = true$

Remark. Into the definition of MultiInstTransition$(node)$ one has to include the dynamic update of the set $Activity(p)$ of all running instances of multiple instances within process instance p, since this set is used for the description of the behavior of end event transitions (in the submachine DeleteAllTokens of EmitResult). It suffices to insert into some submachines some additional updates as follows:

- include Insert$(inst, Activity(proc(t))$ in every place (namely in MultiInstTransition$(node)$) where the start of the execution of a multiple instance $inst$ is described,
- include the update Delete$(inst, Activity(proc(t)))$ where the completion event of an activity instance $inst$ is described (namely in LoopExit for the sequential case and for the parallel case in ComplMultInstExit).

AdHoc Processes *AdHoc* processes are defined in [15, Table 9.14 p.56-57] as subprocesses of type Embedded whose *AdHoc* attribute is set to true. The declared intention is to describe by such processes activities that "are not controlled or sequenced in any particular order" by the activity itself, leaving their control to be "determined by the performers of the activities". Nevertheless an *adHocOrder*ing function is provided to specify either a parallel execution (the default case) or a sequential one.[35]

Notably the definition of when an adhoc activity is *Completed* is left to a monitored predicate *AdHocCompletionCond*ition, which "cannot be defined beforehand" (ibid.p.132) and is required to be "determined by the performes of the activites". Therefore the execution of the rule for an adhoc process continues as long as the *AdHocCompletionCond*ition has not yet become true; there is no further enabledness condition for the subprocesses of an ad hoc processes. As a consequence it is probably implicitly required that the *AdHocCompletionCond*ition becomes true when all the "activities within an AdHoc Embedded Sub-Process", which we denote by a set (parallel case) or list (sequential case) *innerAct*, are *Completed*. Thus the transition rule to describe the behavior of an adhoc activity can be formalized as follows.

AdHocTransition$(node)$ = [**if** $Enabled(in)$ **then**]
  let $t = firingToken(in)$
    [Consume(t, in)]
    [**let** $i = select_{InputSets}(SomeAvail(inputSets(node)))$
      $currInput(node) := i$]
    while not $AdHocCompletionCond(node, t)$
      if $adHocOrder(node) = Parallel$ **then forall** $a \in innerAct(node)$ **do** $a[inputs(i)]$
      if $adHocOrder(node) = Sequential$ **then let**$< a_0, \ldots, a_n >= innerAct(node)$
        foreach $j < n$ **do** $a_j[inputs(i)]$
    seq LoopExit$(node, t)$
      where $Completed(node, t) = AdHocCompletionCond(node, t)$

Remark on completely undefined ad hoc behavior In [15, Sect.10.2.3 p.132] yet another understanding of "the sequence and number of performances" of the inner activities of an adhoc process is stated, namely that "they can be performed in almost (Sic) any order or frequency"

[35] For the description of the parallel case we use the parallel ASM construct **forall**, for the sequential case the **foreach** construct as defined for ASMs in Sect. 10 using **seq**.

and that "The performers determine when activities will start, when they will end, what the next activity will be, and so on". The classification into sequential and parallel *adHocOrder* seems to disappear in this interpretation, in which *any* behavior one can imagine could be inserted. We have difficulties to believe that such a completely non-deterministic understanding is intended as BPMN standard conform. To clarify what the issue is about, we rewrite the transition rule for adhoc processes by explicitly stating that as long as *AdHocCompletionCond* is not yet true, repeatedly a multi-set of inner activities can be chosen and executed until completion. The fact that the choice happens in a non-deterministic manner, which will only be defined by the implementation or at runtime, is made explicit by using the **choose** construct for ASMs (see Sect. 10 for an explanation). We use $A \subseteq_{multi} B$ to denote that A is a multi-set of elements from B.

UnconstrainedAdHocTransition($node$) = [**if** *Enabled*(in) **then**]
  let t = *firingToken*(in)
    [Consume(t, in)]
    [**let** i = $select_{InputSets}$(*SomeAvail*(*inputSets*($node$)))
      currInput($node$) := i]
    while not *AdHocCompletionCond*($node$, t)
      choose $A \subseteq_{multi}$ *innerAct*($node$)
        forall $a \in A$ **do** a[*inputs*(i)]
    seq LoopExit($node$, t)
      where *Completed*($node$, t) = *AdHocCompletionCond*($node$, t)

Many issues remain open with such an interpretation. For example, can an activity within an ad hod embedded subprocess be transactional? Can it be an iteration? What happens if during one execution round for a chosen subset A of embedded activities one of these throws an exception that cannot be caught within the embedded activity itself? Can ad hod subprocesses be nested? If yes, how are exceptions and transactional requirements combined with nesting? Etc.

6.3 Subprocess Nodes

The main role of subprocesses is to represent modularization techniques. Their role in creating an EventContext for exception handling, cancellation and compensation has already been described above when formalizing the behavior of intermediate events that are placed on the boundary of an activity. Their role in showing parallel activities has been dealt with by the description of iterative (in particular adhoc) processes. The normal sequence flow of their inner activities is already formalized by the preceding description of the behavior of tasks, events and gateways, using that subprocess activities in BPMN have the same sequence flow connections as task activities. What remains to be described is their role when calling an activity, which may involve an instantiation and passing data from caller to callee, and when coming back from an activity.

For the discussion of calling and returning from subprocesses we can start from the BPMN Best Practice Normal Form assumption as made for tasks, namely that there is (at most) one *in*coming and (at most) one *out*going arc. For calling a subprocess we can assume that when an arc incoming a subprocess is enabled, the start event of the process if triggered. This stipulation comes up to be part of the definition of the *Triggered* predicate for such start events, where we assume for the token model that the event type is Link and that *startToken* con veys the token information related to this link to the the token created when the subprocess starts. If there is no incoming arc, then the standard stipulation is that the subprocess (if it is not a compensation) is enabled when its parent process is enabled. We can include this into the description of the previous case by considering that there is a special virtual arc in our graph representation that leads from the parent process to each of its (parallel) subprocesses. We have dealt in a similar way with returning from a subprocess via end events, which bring the sequence flow back to the parent process (see the definition of EmitResult for end events in Sect. 5). This is in accordance with the illustrations in [15, Fig.10.14-16 p.108-110] for dealing with start/end events that are attached to the boundary of an expanded subprocess (see also the characteristic example in [15, Fig.10.48 p.127]).

There is not much one can do to formalize instantiation aspects since the standard document leaves most of the details open. For example concerning the instantiation of a process called by a so-called *independent* subprocess it is stated that "The called Process will be instantiated when called but it can be instantiated by other Independent Sub-Process objects (in other diagrams) or by a message from an external source" [15, Sect.9.4.2 p.57]. This does not mean that there is not a certain number of issues to specify to make the subprocess concept clear enough to allow for standard compatible implementations. These issues are related to problems of procedure concepts that are well-known from programming languages. For example, how is the nesting of (recursive?) calls of independent subprocesses dealt with, in particular in relation to the exception handling and the transaction concept? Which binding mechanism for process instances and which parameter passing concept is assumed? Are arbitrary interactions (sharing of data, events, control) between caller and callee allowed? Etc.

7 Related Work

There are two specific papers we know on the definition of a formal semantics of a subset of BPMN. In [17] a Petri net model is developed for a core subset of BPMN which however, due to the well-known lack of high-level concepts in Petri nets, "does not fully deal with: (i) parallel multi-instance activities; (ii) exception handling in the context of subprocesses that are executed multiple times concurrently; and (iii) OR-join gateways. " In [41] it is shown "how a subset of the BPMN can be given a process semantics in Communicating Sequential Processes", starting with a formalization of the BPMN syntax using the Z notation and offering the possibility to use the CSP-based model checker for an analysis of model-checkable properties of business processes written in the formalized subset of BPMN. Both papers present, for a subset of BPMN, technically rather involved models for readers who are knowledgeable in Petri nets respectively CSP, two formalisms one can hardly expect system analysts or business process users to know or to learn. In contrast, the ASM descriptions we have provided here cover every construct of the BPMN standard and use the general form of **if** *Event* **and** *Condition* **then** *Action* rules of Event-Condition-Action systems, which are familiar to most analysts and professionals trained in process-oriented thinking. Since ASMs provide a rigorous meaning to abstract (pseudo-) code, for the verification and validation of properties of ASMs one can adopt every appropriate accurate method, without being restricted to mechanical (theorem proving or model checking) techniques.

The feature-based definition of workflow concepts in this paper is an adaptation of the method used in a similar fashion in [35] for an instructionwise definition, verification and validation of interpreters for Java and the JVM. This method has been developed independently for the definition and validation of software product lines [6], see [5] for the relation between the two methods.

8 Conclusion and Future Work

A widely referenced set of 23 workflow patterns appeared in [37] and was later extended by 20 additional workflow patterns in [33]. The first 23 patterns have been described in various languages, among which BPMN diagrams [15, Sect.10.2], [38], coloured Petri nets [33], an extension of a subset of BPMN [23][36], UML 2.0 in comparison to BPMN [38]. A critical review of the list of these patterns and of their classification appears in [10], where ASM descriptions are used to organize the patterns into instances of eight (four sequential and four parallel) fundamental patterns. It could be interesting to investigate what form of extended BPMN descriptions can be given for the interaction patterns in [3] (formalized by ASMs in [4]), where the communication between multiple processes becomes a major issue, differently from the one-process-view of BPMN diagrams dealt with in this paper, which was motivated by the fact that in BPMN the collaboration between different processes is restricted to what can be expressed in terms of events, message exchange between pools and data exchange between processes.

[36] The extensions are motivated by the desire to capture also the additional 20 workflow patterns

One project of practical interest would be to use the high-level description technique presented in this paper to provide for the forthcoming extension BPMN 2.0 a rigorous description of the semantical consequences of the intended extensions, adapting the abstract BPMN model developed here. For this reason we list at the end of this section some of the themes discussed in this paper where the present BPMN standard asks for more precision or some extension. The scheme for WorkflowTransition is general enough to be easily adaptable to the inclusion of process interaction and resource usage concerns, should such features be considered by the standardization comittee for an inclusion into the planned extension of BPMN to BPMN 2.0, as has been advocated in [39]. To show that this project is feasible we intend to adapt the model developed here for BPMN 1.0 to a refined model for BPMN 1.1.

One can also refine the ASM model for BPMN to an adaptation to the current BPEL version of the ASM model developed in [20,21] for BPEL constructs. The ASM refinement concept can be used to investigate the semantical relation established by the mapping defined in [15, Sect.11] from process design realized in BPMN to its implementation by BPEL executions. In particular one can try to resolve the various issues discussed in [29] and related to the fact that BPMN and BPEL reside at different levels of abstraction and that the mapping must (be proved to) preserve the intended process semantics. This is what in the literature is refered to with the bombastic wording of a "conceptual mismatch" [30] between BPMN and BPEL. One could also use CoreAsm [18,19] for a validation of the models through characteristic workflow patterns.

Another interesting project we would like to see being undertaken is to define an abstract model that either semantically unifies UML 2.0 activity diagrams with BPMN diagrams or allows one to naturally instantiate its concepts to those of the two business process description languages and thus explicitly point to the semantic similarities and differences. This is feasable, it has been done for a comparison of highly complex programming languages like Java and C# in [13] using the corresponding ASM models developed for Java and C# in [35,11].

8.1 List of Some Themes for Reviewing the Current BPMN Standard

We summarize here some of the issues concerning the BPMN standard that have been discussed in the paper, where the reader can find the corresponding background information.

1. Clarify the correlation mechanism for multiple events needed to start a process.
2. Clarify the intended consumption mode for events (in particular timer and messages).
3. Specify the assumptions on the selection of input (see task node section).
4. Clarify the issues related to the interpretation of the classical iteration concepts (e.g. which input is taken for while/loop constructs). In particular clarify the concepts of upstream paths and of parallel paths.
5. Provide a precise definition of activities to be *Completed*, in particular with respect to the iteration concepts for ad hoc processes and MultiInstTransition. Clarify what assumptions are made on the possible simultaneous completion of multiple subprocess instances.
6. Provide a precise definition of interruption and cancel scopes, in particular of the set $Activity(p)$ of running instances of multiple instances within a process instance p.
7. Define the behavioral impact of the concept of (multiple) tokens.
8. Clarify the issues related to the procedural concept of (in particular independent) subprocesses and its relation to the underlying transaction concept.
9. Clarify the (possible nesting of the) exception handling and compensation mechanism (in particular whether it is stack like, as seems to be suggested by [15, Sect.11.13]).
10. Clarify the underlying transaction concept, in particular the interaction between the transaction concepts in the listed non-normative references, namely business transaction protocol, open nested transitions and web services transactions in relation to the group concept of [15, Sect.9.7.4], which is not restricted to one agent executing a pool process.
11. Clarify how undetermined the interpretation of OR-join gateways is intended (specification of the functions $select_{Produce}$ and $select_{Consume}$).

12. Clarify the issues related to the refinement of abstract BPMN concepts to executable versions, in particular their mapping to block-structured BPEL (see [29] for a detailed analysis of problems related to this question).
13. Clarify whether to keep numerous interdefinable constructs or to have a basic set of independent constructs from where other forms can be defined in a standard manner (pattern library).[37]
14. Clarify whether other communication mechanisms than the one in BPEL are allowed.
15. Formulate a best practice discipline for BPMN process diagrams.
16. Add the consideration of resources.
17. Provide richer explicit forms of interaction between processes.

9 Appendix: The BPMN Execution Model in a Nutshell

We summarize here the rules explained in the main text. We do not repeat the auxiliary definitions provided in the main text.

9.1 The Scheduling and Behavioral Rule Schemes

WorkflowTransitionInterpreter =
let $node = select_{Node}(\{n \mid n \in Node$ **and** $Enabled(n)\})$
let $rule = select_{WorkflowTransition}(\{r \mid r \in WorkflowTransition$ **and** $Fireable(r, node)\})$
$rule$

The behavioral rule scheme (form of rules in *WorkflowTransition*):

WorkflowTransition($node$) =
if $EventCond(node)$ **and** $CtlCond(node)$
and $DataCond(node)$ **and** $ResourceCond(node)$ **then**
DataOp($node$)
CtlOp($node$)
EventOp($node$)
ResourceOp($node$)

9.2 Gateway Rules

AndSplitGateTransition($node$) = WorkflowTransition($node$)
where
$CtlCond(node) = Enabled(in)$
CtlOp($node$) =
let $t = firingToken(in)$
Consume(t, in)
ProduceAll($\{(andSplitToken(t, o), o) \mid o \in outArc(node)\}$)
DataOp($node$) = //performed for each selected gate
forall $o \in outArc(node)$ **forall** $i \in assignments(o)$ Assign($to_i, from_i$)

AndJoinGateTransition($node$) = WorkflowTransition($node$)
where
$CtlCond(node) =$ **forall** $in \in inArc(node)$ $Enabled(in)$
CtlOp($node$) =
let $[in_1, \ldots, in_n] = inArc(node)$
let $[t_1, \ldots, t_n] = firingToken(inArc(node))$
ConsumeAll($\{(t_j, in_j)) \mid 1 \leq j \leq n\}$)
Produce($andJoinToken(\{t_1, \ldots, t_n\}), out$)
DataOp($node$) = **forall** $i \in assignments(out)$ Assign($to_i, from_i$)

[37] The problem of redundancy of numerous BPMN constructs has been identified also in [28]. An analogous problem has been identified for UML 2.0 activity diagrams, called "excessive supply of concepts" in [34].

OrSplitGateTransition($node$) =
  let $O = select_{Produce}(node)$ **in** WorkflowTransition($node, O$)
where
  $CtlCond(node) =$ $Enabled(in)$
  CtlOp($node, O$) =
    let $t = firingToken(in)$
      Consume(t, in)
      ProduceAll($\{(orSplitToken(t, o), o) \mid o \in O\}$)
  DataOp($node, O$) = **forall** $o \in O$ **forall** $i \in assignments(o)$ Assign($to_i, from_i$)

Constraints for $select_{Produce}$
$select_{Produce}(node) \neq \emptyset$
$select_{Produce}(node) \subseteq \{out \in outArc(node) \mid OrSplitCond(out)\}$

OrJoinGateTransition($node$) =
  let $I = select_{Consume}(node)$ **in** WorkflowTransition($node, I$)
where
  $CtlCond(node, I) =$ $(I \neq \emptyset$ **and forall** $j \in I$ $Enabled(j))$
  CtlOp($node, I$) =
    Produce($orJoinToken(firingToken(I)), out$)
    ConsumeAll($\{(t_j, in_j) \mid 1 \leq j \leq n\}$) **where**
      $[t_1, \ldots, t_n] = firingToken(I)$
      $[in_1, \ldots, in_n] = I$
  DataOp($node$) = **forall** $i \in assignments(out)$ Assign($to_i, from_i$)

GateTransitionPattern($node$) =
  let $I = select_{Consume}(node)$
  let $O = select_{Produce}(node)$ **in**
    WorkflowTransition($node, I, O$)
where
  $CtlCond(node, I) =$ $(I \neq \emptyset$ **and forall** $in \in I$ $Enabled(in))$
  CtlOp($node, I, O$) =
    ProduceAll($\{(patternToken(firingToken(I), o), o) \mid o \in O\}$)
    ConsumeAll($\{(t_j, in_j) \mid 1 \leq j \leq n\}$) **where**
      $[t_1, \ldots, t_n] = firingToken(I)$
      $[in_1, \ldots, in_n] = I$
  DataOp($node, O$) = **forall** $o \in O$ **forall** $i \in assignments(o)$ Assign($to_i, from_i$)

9.3 Event Rules

StartEventTransition($node$) =
  choose $e \in trigger(node)$ StartEventTransition($node, e$)

StartEventTransition($node, e$) =
  if $Triggered(e)$ **then** Produce($startToken(e), out$)
                ConsumEvent(e)

EndEventTransition($node$) =
  if $Enabled(in)$ **then**
    Consume($firingToken(in), in$)
    EmitResult($firingToken(in), res(node), node$)

```
  EmitResult(t, result, node) =
    if type(result) = Message then Send(mssg(node, t))
    if type(result) ∈ {Error, Cancel, Compensation} then
      Triggered(targetIntermEv(result, node)) := true // trigger intermediate event
      Insert(exc(t), excType(targetIntermEvNode(result, node))))
    if type(result) = Cancel then
      Callback(mssg(cancel, exc(t), node), listener(cancel, node))
    if type(result) = Link then Produce(linkToken(result), result)
    if type(result) = Terminate then DeleteAllTokens(process(t))
    if type(result) = None and IsSubprocessEnd(node) then
      Produce(returnToken(targetArc(node), t), targetArc(node))
    if type(result) = Multiple then
      forall r ∈ MultipleResult(node) EmitResult(t, r, node)


      Callback(m, L) = forall l ∈ L Send(m, l)
      DeleteAllTokens(p) = forall act ∈ Activity(p)
        forall a ∈ inArc(act) forall t ∈ TokenSet(p) Empty(token(a, t))

IntermEventTransition(node) =
  choose e ∈ trigger(node) IntermEventTransition(node, e)

IntermEventTransition(node, e) =
  if Triggered(e) then
    if not BoundaryEv(e) then
      if Enabled(in) then let t = firingToken(in)
        ConsumEvent(e)
        Consume(t, in)
        if type(e) = Link then Produce(linkToken(link), link)
        if type(e) = None then Produce(t, out)
        if type(e) = Message then
          if NormalFlowCont(mssg(node), process(t))
            then Produce(t, out)
            else Throw(exc(mssg(node)), targetIntermEv(node))
        if type(e) = Timer then Produce(timerToken(t), out)
        if type(e) ∈ {Error, Compensation, Rule} then Throw(e, targetIntermEv(e))
    if BoundaryEv(e) then
      if active(targetAct(e)) then
        ConsumEvent(e)
        if type(e) = Timer then Insert(timerEv(e), excType(node))
        if type(e) = Rule then Insert(ruleEv(e), excType(node))
        if type(e) = Message then Insert(mssgEv(e), excType(node))
        if type(e) = Cancel then choose exc ∈ excType(node) in
          if Completed(Cancellation(e, exc)) then Produce(excToken(e, exc), out)
        else TryToCatch(e, node)
where
  TryToCatch(ev, node) =
    if ExcMatch(ev) then Produce(out(ev))
      else TrytoCatch(ev, targetIntermEv(node, ev))
  Completed(Cancellation(e)) =
    RolledBack(targetAct(e)) and Completed(Compensation(targetAct(e)))
```

9.4 Activity Rules

TaskTransition($task$) = [**if** *Enabled*(in) **then**]
  if *ReadyForExec*($task$) **then let** t = *firingToken*(in)
    [Consume(t, in)]
    let i = $select_{InputSets}$(*SomeAvail*(*inputSets*($task$)))
      Exec($task$, *inputs*(i))
      currInput($task$) := i
  [**seq**
    if *Completed*($task$, t) **then**
      [ProduceOutput(*outputSets*($task$), *currInput*($task$))]
      [Produce(*taskToken*($task$, t), *out*)]]
where
  ProduceOutput(*outputSets*(t), i) =
    choose $o \in$ *outputSets*(t) **with** *Defined*(*outputs*(o)) **and** *IORules*(t)(o, i) = *true*
      Emit(*outputs*(o))

ReadyForExec(t) =

$$\begin{cases} \textit{SomeAvail}(\textit{inputSets}(t)) & \textbf{if } \textit{type}(t) \in \{\textit{Service}, \textit{User}\} \\ \textit{Arrived}(\textit{mssg}(t))\ [\textbf{and } \textit{Instantiate}(t)] & \textbf{if } \textit{type}(t) = \textit{Receive} \\ \textit{true} & \textbf{if } \textit{type}(t) \in \{\textit{Send}, \textit{Script}, \textit{Manual}, \textit{Reference}\} \end{cases}$$

Exec(t, i) =

$$\begin{cases} \textsc{Send}(\textit{inMssg}(t)) & \textbf{if } \textit{type}(t) \in \{\textit{Service}, \textit{User}\} \\ \textsc{Receive}(\textit{mssg}(t)) & \textbf{if } \textit{type}(t) = \textit{Receive} \\ \textsc{Send}(\textit{mssg}(t)) & \textbf{if } \textit{type}(t) \in \{\textit{Send}\} \\ \textsc{Call}(\textit{performer}(\textit{action}(t,i)), \textit{action}(t,i)) & \textbf{if } \textit{type}(t) \in \{\textit{Script}, \textit{Manual}\} \\ \textsc{Exec}(\textit{taskRef}(t), i) & \textbf{if } \textit{type}(t) = \textit{Reference} \\ \textbf{skip} & \textbf{if } \textit{type}(t) = \textit{None} \end{cases}$$

LoopTransition($node$) = [**if** *Enabled*(in) **then**]
  let t = *firingToken*(in)
    LoopEntry($node$, t)
    seq
      if *testTime*($node$) = *before* **then**
        while *loopCond*($node$, t) LoopBody($node$, t)
      if *testTime*($node$) = *after* **then**
        until *loopCond*($node$, t) LoopBody($node$, t)
    [**seq** LoopExit($node$, t)]
where
  LoopBody(n, t) =
    loopCounter($node$, t) := *loopCounter*($node$, t) + 1
    iterBody($node$, *loopToken*(t, *loopCounter*($node$, t) + 1)[, *inputs*(*currInput*($node$))])
  LoopEntry(n, t) =
    loopCounter(n, t) := 0
    [Consume(t, in)]
    [*currInput*(n) := $select_{InputSets}$(*SomeAvail*(*inputSets*(n)))]
  LoopExit(n, t) =
    if *Completed*(n, t) **then**
      [ProduceOutput(*outputSets*(n), *currInput*(n))]
      [Produce(*loopExitToken*(t, *loopCounter*(n, t)), *out*)]
  Completed(n, t) = *LoopCompleted*(n, t) **if** $n \in$ *Loop*(t)

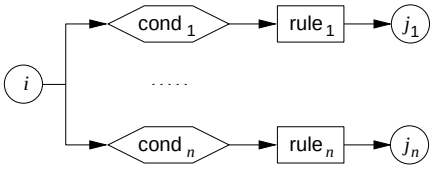
```
MultiInstTransition(node) = [if Enabled(in) then]
  let t = firingToken(in)
    LoopEntry(node, t)
    seq
      if miOrdering(node) = Sequential then
        foreach i ≤ miNumber(node)
          loopCounter(node, t) := loopCounter(node, t) + 1
          iterBody(node, miToken(t, i)[, inputs(currInput(node))])
        seq LoopExit(node, t)
      if miOrdering(node) = Parallel then
        forall i ≤ miNumber(node)
          Start(iterBody(node, miToken(t, i)[, inputs(currInput(node))]))
        seq
          if miFlowCond = All then
            if Completed(node, t) then LoopExit(node, t)
          if miFlowCond = None then EveryMultInstExit(node, t)
          if miFlowCond = One then OneMultInstExit(node, t)
          if miFlowCond = Complex then ComplMultInstExit(node, t)
  where
    Completed(n, t) = forall i ≤ miNumber(n) Completed(iterBody(n, miToken(t, i)[...]))
    ComplMultInstExit(n, t) = // for miOrdering(n) = Parallel
      AlreadyCompleted := ∅ // initially no instance is completed
      seq
        while AlreadyCompleted ≠ {i | i ≤ miNumber(n)} do
          if NewCompleted(n, t) ≠ ∅ then
            if | AlreadyCompleted |< tokenNo(complexMiFlowCond)
            then
              if TokenTime(complexMiFlowCond) then
                let i_0 = select_NewCompleted in
                  Produce(miExitToken(t, i_0), out)
                  Insert(i_0, AlreadyCompleted)
                [ProduceOutput(outputSets(n), currInput(n))]
            else forall i ∈ NewCompleted(n, t) Insert(i, AlreadyCompleted)
    NewCompleted(n, t) = {i ≤ miNumber(n) |
      Completed(iterBody(n, miToken(t, i)[...]))
      and i ∉ AlreadyCompleted}
    EveryMultInstExit(n, t) = ComplMultInstExit(n, t)
      where
        tokenNo(complexMiFlowCond) =| {i | i ≤ miNumber(n)} |
        TokenTime(complexMiFlowCond) = true
    OneMultInstExit(n, t) = ComplMultInstExit(n, t)
      where
        tokenNo(complexMiFlowCond) = 1
        TokenTime(complexMiFlowCond) = true

UnconstrainedAdHocTransition(node) = [if Enabled(in) then]
  let t = firingToken(in)
    [Consume(t, in)]
    [let i = select_InputSets(SomeAvail(inputSets(node)))
      currInput(node) := i]
    while not AdHocCompletionCond(node, t)
      choose A ⊆_multi innerAct(node)
        forall a ∈ A do a[inputs(i)]
    seq LoopExit(node, t)
```

```
AdHocTransition(node) = [if Enabled(in) then]
  let t = firingToken(in)
    [Consume(t, in)]
    [let i = select_InputSets(SomeAvail(inputSets(node)))
      currInput(node) := i]
    while not AdHocCompletionCond(node, t)
      if adHocOrder(node) = Parallel then forall a ∈ innerAct(node) do a[inputs(i)]
      if adHocOrder(node) = Sequential then let< a_0, ..., a_n >= innerAct(node)
        foreach j < n  do a_j[inputs(i)]
    seq LoopExit(node, t)
      where Completed(node, t) = AdHocCompletionCond(node, t)
```

10 Appendix: ASMs in a nutshell

The ASM method for high-level system design and analysis (see the AsmBook [12]) comes with a simple mathematical foundation for its three constituents: the notion of *ASM*, the concept of *ASM ground model* and the notion of *ASM refinement*. For an understanding of this paper only the concept of ASM is needed. For the concept of ASM ground model (read: mathematical system blueprint) and ASM refinement see [9].

10.1 ASMs = FSMs with arbitrary locations




```
if ctl_state = i then
  if cond_1 then
    rule_1
    ctl_state := j_1
      ...
  if cond_n then
    rule_n
    ctl_state := j_n
```

Fig. 1. Viewing FSM instructions as control state ASM rules

The instructions of a Finite State Machine (FSM) program are pictorially depicted in Fig. 1, where $i, j_1, \ldots, j_n$ are internal (control) states, $cond_\nu$ (for $1 \leq \nu \leq n$) represents the input condition $in = a_\nu$ (reading input a_ν) and $rule_\nu$ the output action $out := b_\nu$ (yielding output b_ν), which goes together with the *ctl_state* update to j_ν. Control state ASMs have the same form of programs and the same notion of run, but the underlying notion of state is extended from the following three locations:

- a single internal *ctl_state* that assumes values in a not furthermore structured finite set
- two input and output locations *in*, *out* that assume values in a finite alphabet

to a *set of possibly parameterized locations holding values of whatever types*. Any desired level of abstraction can be achieved by permitting to hold values of arbitrary complexity, whether atomic or structured: objects, sets, lists, tables, trees, graphs, whatever comes natural at the considered level of abstraction. As a consequence an FSM step, consisting of the simultaneous update of the *ctl_state* and of the *out*put location, is turned into an ASM step consisting of the simultaneous update of a set of locations, namely via multiple assignments of the form $loc(x_1, \ldots, x_n) := val$, yielding a new ASM state.

This simple change of view of what a state is yields machines whose states can be arbitrary *multisorted structures*, i.e. domains of whatever objects coming with predicates (attributes) and

functions defined on them, structures programmers nowadays are used to from object-oriented programming. In fact such a memory structure is easily obtained from the flat location view of abstract machine memory by grouping subsets of data into tables (arrays), via an association of a value to each table entry $(f, (a_1, \ldots, a_n))$. Here f plays the role of the name of the table, the sequence $(a_1, \ldots, a_n)$ the role of a table entry, $f(a_1, \ldots, a_n)$ denotes the value currently contained in the location $(f, (a_1, \ldots, a_n))$. Such a table represents an array variable f of dimension n, which can be viewed as the current interpretation of an n-ary "dynamic" function or predicate (boolean-valued function). This allows one to structure an ASM state as a set of tables and thus as a multisorted structure in the sense of mathematics.

In accordance with the extension of unstructured FSM control states to ASM states representing arbitrarily rich structures, the FSM-input *cond*ition is extended to arbitrary ASM-state expressions, namely formulae in the signature of the ASM states. They are called *guards* since they determine whether the updates they are guarding are executed.[38] In addition, the usual non-deterministic interpretation, in case more than one FSM-instruction can be executed, is replaced by the parallel interpretation that in each ASM state, the machine executes simultaneously all the updates which are guarded by a condition that is true in this state. This *synchronous parallelism*, which yields a clear concept of *locally described global state change*, helps to abstract for high-level modeling from irrelevant sequentiality (read: an ordering of actions that are independent of each other in the intended design) and supports refinements to parallel or distributed implementations.

Including in Fig. 1 $ctl\_state = i$ into the guard and $ctl\_state := j$ into the multiple assignments of the rules, we obtain the definition of a *basic ASM* as a set of instructions of the following form, called ASM *rules* to stress the distinction between the parallel execution model for basic ASMs and the sequential single-instruction-execution model for traditional programs:

if *cond* **then** *Updates*

where *Updates* stands for a set of *function updates* $f(t_1, \ldots, f_n) := t$ built from expressions t_i, t and an n-ary function symbol f. The notion of run is the same as for FSMs and for transition systems in general, taking into account the synchronous parallel interpretation.[39] Extending the notion of mono-agent sequential runs to asynchronous (also called partially ordered) multi-agent runs turns FSMs into globally asynchronous, locally synchronous Codesign-FSMs [27] and similarly basic ASMs into *asynchronous ASMs* (see [12, Ch.6.1] for a detailed definition).

The synchronous parallelism (over a finite number of rules each with a finite number of to-be-updated locations of basic ASMs) is often further extended by a synchronization over arbitrary many objects in a given *Set*, which satisfy a certain (possibly runtime) *Property*:

forall $x[\in Set][$**with** $Property(x)]$ **do**
  $rule(x)$

standing for the execution of *rule* for every object x, which is element of *Set* and satisfies *Property*. Sometimes we omit the key word **do**. The parts $\in Set$ and **with** $Property(x)$ are optional and therefore written in square brackets.

Where the sequential execution of first M followed by N is needed we denote it by M **seq** N, see [12] for a natural definition in the context of the synchronous parallelism of ASMs. We sometimes use also the following abbreviation for iterated sequential execution, where n is an integer-valued location:

foreach $i \leq n$ **do** $rule(i) =$
  $rule(1)$ **seq** $rule(2)$ **seq** $\ldots$ **seq** $rule(n)$

[38] For the special role of *in/out*put locations see below the classification of locations.

[39] More precisely: to execute one step of an ASM in a given state S determine all the fireable rules in S (s.t. *cond* is true in S), compute all expressions t_i, t in S occuring in the updates $f(t_1, \ldots, t_n) := t$ of those rules and then perform simultaneously all these location updates if they are consistent. In the case of inconsistency, the run is considered as interrupted if no other stipulation is made, like calling an exception handling procedure or choosing a compatible update set.

ASM Modules Standard module concepts can be adopted to syntactically structure large ASMs, where the module interface for the communication with other modules names the ASMs which are imported from other modules or exported to other modules. We limit ourselves here to consider an ASM module as a pair consisting of *Header* and *Body*. A module header consists of the name of the module, its (possibly empty) import and export clauses, and its signature. As explained above, the signature of a module determines its notion of state and thus contains all the basic functions occurring in the module and all the functions which appear in the parameters of any of the imported modules. The body of an ASM module consists of declarations (definitions) of functions and rules. An ASM is then a module together with an optional characterization of the class of initial states and with a compulsory additional (the main) rule. Executing an ASM means executing its main rule. When the context is clear enough to avoid any confusion, we sometimes speak of an ASM when what is really meant is an ASM module, a collection of named rules, without a main rule.

ASM Classification of Locations and Functions The ASM method imposes no a priori restriction neither on the abstraction level nor on the complexity nor on the means of definition of the functions used to compute the arguments and the new value denoted by t_i, t in function updates. In support of the principles of separation of concerns, information hiding, data abstraction, modularization and stepwise refinement, the ASM method exploits, however, the following distinctions reflecting the different roles these functions (and more generally locations) can assume in a given machine, as illustrated by Figure 2 and extending the different roles of *in*, *out*, *ctl_state* in FSMs.

A function f is classified as being of a given type if in every state, every location $(f, (a_1, \ldots, a_n))$ consisting of the function name f and an argument $(a_1, \ldots, a_n)$ is of this type, for every argument $(a_1, \ldots, a_n)$ the function f can take in this state.

Semantically speaking, the major distinction is between static and dynamic locations. Static locations are locations whose values do not depend on the dynamics of states and can be determined by any form of satisfactory state-independent (e.g. equational or axiomatic) definitions. The further classification of dynamic locations with respect to a given machine M supports to distinguish between the roles different 'agents' (e.g. the system and its environment) play in using (providing or updating the values of) dynamic locations. It is defined as follows:

- *controlled* locations are readable and writable by M,
- *monitored* locations are for M only readable, but they may be writable by some other machine,
- *output* locations are by M only writable, but they may be readable by some other machine,

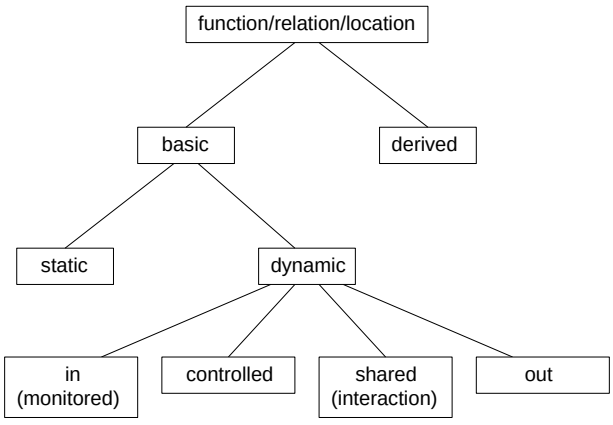


Fig. 2. Classification of ASM functions, relations, locations

- *shared* locations are readable/writable by M as well as by some other machine, so that a protocol will be needed to guarantee the consistency of writing.

Monitored and shared locations represent an abstract mechanism to specify communication types between different agents, each executing a basic ASM. *Derived* locations are those whose definition in terms of locations declared as basic is fixed and may be given separately, e.g. in some other part ("module" or "class") of the system to be built. The distinction of derived from basic locations implies that a derived location can in particular not be updated by any rule of the considered machine. It represents the input-output behavior performed by an independent computation. For details see the AsmBook [12, Ch.2.2.3] from where Figure 2 is taken.

A particularly important class of monitored locations are selection locations, which are frequently used to abstractly describe scheduling mechanisms. The following notation makes the inherent non-determinism explicit in case one does not want to commit to a particular selection scheme.

choose $x[\in \mathit{Set}][\textbf{with } \mathit{Property}(x)][\textbf{do}]$
  $\mathit{rule}(x)$

This stands for the ASM executing $\mathit{rule}(x)$ for some element x, which is arbitrarily chosen among those which are element of *Set* and satisfy the selection criterion *Property*. Sometimes we omit the key word **do**. The parts $\in \mathit{Set}$ and **with** $\mathit{Property}(x)$ are optional.

We freely use common notations with their usual meaning, like **let** $x = t$ **in** R, **if** *cond* **then** R **else** S, list operations like $\mathit{zip}((x_i)_i, (y_i)_i) = (x_i, y_i)_i$, etc.

Non-determinism, Selection and Scheduling Functions It is adequate to use the **choose** construct of ASMs if one wants to leave it completely unspecified who is performing the choice and based upon which selection criterion. The only thing the semantics of this operator guarantees is that each time one element of the set of objects to choose from will be chosen. Different instances of a selection, even for the same set in the same state, may provide the same element or maybe not. If one wants to further analyze variations of the type of choices and of who is performing them, one better declares a *select*ion function, to select an element from the underlying set of *Cand*idates, and writes instead of **choose** $c \in \mathit{Cand}$ **do** $R(c)$ as follows, where R is any ASM rule:

let $c = \mathit{select}(\mathit{Cand})$ **in** $R(c)$

The functionality of *select* guarantees that exactly one element is chosen. The **let** construct guarantees that the choice is fixed in the binding range of the **let**. Declaring such a function as dynamic guarantees that the selection function applied to the same set in different states may return different elements. Declaring such a function as controlled or monitored provides different ownership schemes. Naming these selection functions allows the designer in particular to analyze and play with variations of the selection mechanisms due to different interpretations of the functions.

Acknowledgement We thank the following colleagues for their critical remarks on preliminary versions of this paper: M. Altenhofen, B. Koblinger, M. Momotko, A. Nowack.

Final version to appear in *Advances in Software Engineering*, Springer LNCS 5316 (2008)

References

1. OMG Unified Modeling Language superstructure (final adopted specification, version 2.0), 2003. www.omg.org.
2. Web Services Business Process Execution Language version 2.0. OASIS Standard, April 11 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
3. A.Barros, M.Dumas, and A. Hofstede. Service interaction patterns. In *Proc.3rd International Conference on Business Process Management (BPM2005)*, LNCS, pages 302–318, Nancy, 2005. Springer.
4. A. Barros and E. Börger. A compositional framework for service interaction patterns and communication flows. In K.-K. Lau and R. Banach, editors, *Formal Methods and Software Engineering. Proc. 7th International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *LNCS*, pages 5–35. Springer, 2005.

5. D. Batory and E. Börger. Modularizing theorems for software product lines: The Jbook case study. *J. Universal Computer Science*, Special ASM Issue, 2008. Extended abstract "Coupling Design and Verification in Software Product Lines" of FoIKS 2008 Keynote in: S. Hartmann and G. Kern-Isberner (Eds): FoIKS 2008 (Proc. of *The Fifth International Symposium on Foundations of Information and Knowledge Systems*), Springer LNCS 4932, p.1–4, 2008.
6. D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. In *ACM TOSEM*. ASM, October 1992.
7. E. Börger. High-level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, volume 1641 of *Lecture Notes in Computer Science*, pages 1–43. Springer-Verlag, 1999.
8. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
9. E. Börger. Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. *Formal Aspects of Computing*, 19:225–241, 2007.
10. E. Börger. Modeling workflow patterns from first principles. In V. C. S. C. Parent, K.-D. Schewe and B. Thalheim, editors, *Conceptual Modeling–ER 2007*, volume 4801 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2007.
11. E. Börger, G. Fruja, V. Gervasi, and R. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2–3):235–284, 2005.
12. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
13. E. Börger and R. F. Stärk. Exploiting Abstraction for Specification Reuse. The Java/C# Case Study. In M. Bonsangue, editor, *Formal Methods for Components and Objects: Second International Symposium (FMCO 2003 Leiden)*, volume 3188 of *Lecture Notes in Computer Science (ISBN 3-540-22942-6, ISSN 0302-9743)*, pages 42–76. Springer, 2004. .
14. E. Börger and B. Thalheim. On defining the behavior of OR-joins in business process models. In preparation
15. BPMI.org. Business Process Modeling Notation Specification v.1.0. dtc/2006-02-01 at http://www.omg.org/technology/documents/spec_catalog.htm, 2006.
16. BPMI.org. Business Process Modeling Notation Specification v.1.1. http://www.omg.org/spec/BPMN/1.1/PDF, 2008. formal/2008-01-17.
17. R. M. Dijkman, M. Dumas, and C. Ouyang. Formal semantics and analysis of BPMN process models using Petri nets. Technical Report 7115, Queensland University of Technology, Brisbane, 2007.
18. R. Farahbod et al. *The CoreASM Project*. http://www.coreasm.org.
19. R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An Extensible ASM Execution Engine. *Fundamenta Informaticae XXI*, 2006.
20. R. Farahbod, U. Glässer, and M. Vajihollahi. Specification and validation of the Business Process Execution Language for web services. In W. Zimmermann and B. Thalheim, editors, *Abstract Sate Machines 2004*, volume 3052 of *Lecture Notes in Computer Science*, pages 78–94. Springer-Verlag, 2004.
21. R. Farahbod, U. Glässer, and M. Vajihollahi. An Abstract Machine Architecture for Web Service Based Business Process Management. *Int. J. Business Process Integration and Management*, 1(4):279–291, 2006.
22. J. Freund. BPM-software–2008. www.comunda.com, Berlin (Germany), 2008.
23. A. Grosskopf. xBPMN. Formal control flow specification of a BPMN based process execution language. Master's thesis, HPI at Universität Potsdam, July 2007. pages 1-142.
24. V. Gruhn and R. Laue. How style checking can improve business process models. In *Proc. 8th International Conference on Enterprise Information Systems (ICEIS 2006)*, Paphos (Cyprus), May 2006.
25. V. Gruhn and R. Laue. What business process modelers can learn from programmers. *Science of Computer Programming*, 65:4–13, 2007.
26. D. E. Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information at Stanford/ California, 1992.
27. L. Lavagno, A. Sangiovanni-Vincentelli, and E. M. Sentovitch. Models of computation for system design. In E. Börger, editor, *Architecture Design and Validation Methods*, pages 243–295. Springer-Verlag, 2000.
28. M. Listiani. Review on business process modeling notation. Master's thesis, Institute of Telematics of Hamburg University of Technology, July 2008.
29. C. Ouyang, M. Dumas, W. M. P. van der Aalst, and A. H. M. Hofstede. From business process models to process-oriented software systems: The BPMN to BPEL way. Technical Report 06-27, BPMcenter, http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/, 2006.

30. J. Recker and J. Mendling. *Research Issues in Systems Analysis and Design, Databases and Software Development*, chapter Lost in Business Process Model Translations: How a Structured Approach helps to Identify Conceptual Mismatch, pages 227–259. IGI Publishing, Hershey, Pennsylvania, 2007.
31. N. Russel, A. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow data patterns. BPM-04-01 at BPMcenter.org, 2004.
32. N. Russel, A. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow resource patterns. BPM-04-07 at BPMcenter.org, 2004.
33. N. Russel, A. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. BPM-06-22 at http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/, July 2006.
34. T. Schattkowsky and A. Förster. On the pitfalls of UML 2 activity modeling. In *International Workshop on Modeling in Software Engineering (MISE'07)*. IEEE, 2007.
35. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
36. H. Störrle and J. H. Hausman. Towards a formal semantics of UML 2.0 activities. In *Proc. Software Engineering 2005*, 2005.
37. W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.
38. S. A. White. Process modeling notations and workflow patterns. pbmn.org/Documents, 2007 (download September).
39. P. Wohed, W. M. P. van der Aalst, M. Dumas, A. ter Hofstede, and N. Russel. On the suitability of BPMN for business process modelling. submitted to the 4th Int. Conf. on Business Process Management, 2006.
40. P. Wohed, W. M. P. van der Aalst, M. Dumas, A. ter Hofstede, and N. Russel. Pattern-based analysis of BPMN - an extensive evaluation of the control-flow, the data and the resource perspectives (revised version). BPM-06-17 at BPMcenter.org, 2006.
41. P. Y. H. Wong and J. Gibbons. A process semantics fo BPMN. Preprint Oxford University Computing Lab URL: http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/bpmn_extended.pdf, July 2007.
42. M. Wynn, D. Edmond, W. van der Aalst, and A. ter Hofstede. Achieving a general, formal and decidable approach to the OR-join in workflow using reset nets. In *Application and Theory of Petri Nets 2005*, volume 3536 of *LNCS*, pages 423–443. Springer, 2005.