BPMN and Beyond
Business process modelling notation, workflow modelling, functionality of systems

Collection of Recent Papers

Bernhard Thalheim
Christian-Albrechts-University Kiel, Department of Computer Science, 24098 Kiel, Germany
thalheim@is.informatik.uni-kiel.de

   published in [BT08a]
   related papers: [Tha07a,Tha08,ST06,ST04b,ST05,FRT05]
   see also my website under miscellaneous or Verschiedenes: talks 2007/2008

2. On defining the behavior of OR-joins in business process models
   published in [BST09]
   see also my website under miscellaneous or Verschiedenes: talks 2007/2008

3. Modeling Workflows, Interaction Patterns, Web Services and Business Processes:
   The ASM-Based Approach
   published in [BT08b]
   related papers: [KSTZ03,ST04a]
   see also my website under miscellaneous or Verschiedenes: talks 2007/2008

4. ASM Foundations of Database Management
   published in [ST08a]
   related papers: [PT03,BZT03,Tha01,BZKST05,WST08]
   see also my website under miscellaneous or Verschiedenes: talks 2007/2008

5. The Enhanced Entity-Relationship Model
   published in [Tha09]
   related papers: [Tha00,Tha07a,ST08b,ST08a,Tha07b,DMT07,MDT04]
   see also my website under miscellaneous or Verschiedenes: talks 2007/2008

6. Generalisation and specialisation
   published in [Tha09]
   related papers: [Tha00,BT07]
   see also my website under miscellaneous or Verschiedenes: talks 2007/2008

7. Abstraction
   published in [Tha09]
   related papers: [Tha00]
   see also my website under miscellaneous or Verschiedenes: talks 2007/2008
References


A Method for Verifiable and Validatable Business Process Modeling

Egon Börger\textsuperscript{1} and Bernhard Thalheim\textsuperscript{2}

\textsuperscript{1} Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy boerger@di.unipi.it
\textsuperscript{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.\textsuperscript{1}

1 Introduction

Various standardization efforts have been undertaken to reduce the fragmentation of business process modeling notations and tools, most notably BPMN \textsuperscript{15}, UML 2.0 activity diagrams \textsuperscript{1} and BPEL \textsuperscript{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 \textsuperscript{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

\textsuperscript{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. 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.

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, 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 between 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 \cite{37,33} separately also about data \cite{31} and resource \cite{32} patterns (see also \cite{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 accessibility 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.\footnote{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.}

External control comes through input, e.g. messages, timers, trigger signals or conditions. This is about so-called monitored locations\footnote{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.}, 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(\textit{arc}), target(\textit{arc}) for source and target node of an \textit{arc}, pred(\textit{node}) for the (possibly
ordered) set of source nodes of arcs that have the given node as target node, \textit{inArc(node)} for the set of arcs with node as target node, similarly \textit{succ(node)} for the (possibly ordered) set of target nodes of arcs that have the given node as source node, \textit{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 \textit{Enabled} in a state (not to be confused with the omonymous \textit{Enabled}ness predicate for arcs) if at least one of its associated rules is \textit{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 \textit{selectNode} and \textit{selectWorkflowTransition} defined over the sets \textit{Node} of nodes and \textit{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.

\textbf{WorkflowTransitionInterpreter} =
\begin{align*}
\text{let } & \text{node} = \text{selectNode}\left\{ n \mid n \in \text{Node} \text{ and } \text{Enabled}(n) \right\} \\
\text{let } & \text{rule} = \text{selectWorkflowTransition}\left\{ r \mid r \in \text{WorkflowTransition} \text{ and } \text{Fireable}(r, \text{node}) \right\} \\
\text{rule}
\end{align*}

Thus for every workflow construct associated to a node, its behavioral meaning is expressed by a guarded transition rule \textit{WorkflowTransition(node)} \in \textit{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.\footnote{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.}

\textbf{WorkflowTransition(node)} =
\begin{align*}
\text{if } & \text{EventCond(node)} \text{ and } \text{CtlCond(node)} \\
& \text{and } \text{DataCond(node)} \text{ and } \text{ResourceCond(node)} \text{ then} \\
& \text{DataOp(node)} \\
& \text{CtlOp(node)} \\
& \text{EventOp(node)} \\
& \text{ResourceOp(node)}
\end{align*}

\textit{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 \cite{37,33} from a few ASM workflow patterns see \cite{10}.\footnote{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, referring 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 $\text{graph}(\text{process})$ 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:

$$\text{Node} = \text{Activity} \cup \text{Event} \cup \text{Gateway}$$

To define the behavioral meaning of each BPMN flow object one may find in a node, we instantiate in the WorkflowTransition$(\text{node})$ 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$(\text{node})$. The instantiation of EventCond$(\text{node})$ interprets the cause (‘trigger’) of an event happening at the node; the instantiation of EVENTOP$(\text{node})$ interprets 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$(\text{node})$ and the submachines CtlOP$(\text{node})$ of WorkflowTransition$(\text{node})$. Accompanying instantiations of DataCond$(\text{node})$ and DataOP$(\text{node})$ 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$(\text{node})$ and the submachines DataOP$(\text{node})$ of WorkflowTransition$(\text{node})$. For so-called non-atomic activities, which involve subprocesses and possibly iterations over them, we will see a simultaneous instantiation also of the CtlCond$(\text{node})$ guards and of the submachines CtlOP$(\text{node})$ 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 selectNode and selectWorkflowTransition) and b) WorkflowTransition$(\text{node})$ 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$(\text{node})$ scheme for each concrete node of the diagram. This implies instantiations of the diagram related abstractions used by WorkflowTransition$(\text{node})$, 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.
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 $\text{CtlCond}(\text{node})$ in the guard and $\text{CtlOp}(\text{node})$ in the body of BPMN instances of rules of form $\text{WorkflowTransition}(\text{node})$.

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). In the following, unless otherwise stated, by arc we always mean a sequence flow arc and use $\text{Arc}$ 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 $\text{in}$ resp. $\text{out}$ instead of $\text{inArc}(\text{node}) = \{\text{in}\}$ resp. $\text{outArc}(\text{node}) = \{\text{out}\}$.

### 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 $\text{Token}$—to arcs, using a dynamic function $\text{token}(\text{arc})$. 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 $\text{token}_{\pi \text{in}}$ to represent the current token marking in the process diagram instance of the process instance $\pi$ a token belongs to, so that $\text{token}_{\pi \text{in}}(\text{arc})$ 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.

$$\text{token} : \text{Arc} \rightarrow \text{Multiset(\text{Token})}$$

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 $\text{Type}$ 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 \( \text{token}_{pi}(in) \),
denoted \( \mid \text{token}_{pi}(in) \mid \) is at least the quantity \( \text{inQty}(in) \) 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 \( \text{Enabled} \).

\[
\text{Enabled}(in) = (\mid \text{token}_{pi}(in) \mid \geq \text{inQty}(in) \text{ for some } pi)
\]

Correspondingly the control operation \( \text{CtlOp} \) of a workflow usually consists of two parts,
one describing how many tokens are \( \text{Consumed} \) on which incoming arcs and one describing how
many tokens are \( \text{Produced} \) on which outgoing arcs, indicated by using an analogous abstract
function \( \text{outQty} \). 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.

\[
\begin{align*}
\text{CONSUME}(t, in) & = \text{DELETE}(t, \text{inQty}(in), \text{token}(in)) \\
\text{PRODUCE}(t, out) & = \text{INSERT}(t, \text{outQty}(out), \text{token}(out)) \\
\text{PASS}(t, in, out) & = \\
& \quad \text{CONSUME}(t, in) \\
& \quad \text{PRODUCE}(t, out)
\end{align*}
\]

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
\( \text{firingToken}(A) \), which is assumed to select for each element \( a \) of an ordered set \( A \) of incoming arcs
tokens from \( \text{token}_{pi}(a) \) that enable \( a \), all belonging to the same process instance \( pi \) and ready to
be \( \text{Consumed} \). For the sake of exposition we make the usual assumption that \( \text{inQty}(in) = 1 \), so
that we can use the following sequence notation:

\[
\text{firingToken}([a_1, \ldots, a_n]) = [t_1, \ldots, t_n]
\]

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

If one considers, as seems to be very often the case, only (multiple occurrences of) indistin-
guishable 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:

\[
\begin{align*}
\text{CONSUME}(in) & = \text{DELETE}(\text{inQty}(in), \text{token}(in)) \\
\text{PRODUCE}(out) & = \text{INSERT}(\text{outQty}(out), \text{token}(out)) \\
\text{CONSUMEAll}(X) & = \text{forall } x \in X \text{ CONSUME}(x) \\
\text{PRODUCEAll}(Y) & = \text{forall } y \in Y \text{ PRODUCE}(y)
\end{align*}
\]

Remark. This use of macros allows one to adapt the abstract token model to different instan-
tiations 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 \( \text{positioned} \), as seems to be popular
in implementations, then it suffices to refine the macro for \( \text{Passing a token } t \) from \( in \) to \( out \) by
updating the second token component, namely from its current \( \text{position value } in \) to its new value
\( out \):

\[
\text{PASS}(t, in, out) = (\text{pos}(t) := out)
\]

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

\(^{11}\) The function \( \text{inQty} \) generalizes the \( \text{startQuantity} \) 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 selectNode and selectWorkflowTransition) 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.
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 \( \text{WorkflowTransition(node)} \) for gateway nodes below by corresponding \( \text{EventCond(node)} \) and \( \text{DataCond(node)} \) 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 \( \text{CtlCond} \) 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 how 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 occurrences 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:

\[
\text{ANDSplitGateTransition}(node) = \text{WorkflowTransition}(node)
\]

where

\[
\begin{align*}
CtlCond(node) & = Enabled(in) \\
CtlOp(node) & = \\
& \text{let } t = \text{firingToken}(in) \\
& \text{CONSUME}(t, in) \\
& \text{PRODUCE}(\{\text{andSplitToken}(t, o) | o \in outArc(node)\}) \\
DataOp(node) & = //\text{performed for each selected gate} \\
& \text{forall } o \in outArc(node) \text{ forall } i \in assignments(o) \text{ ASSIGN}(to_i, from_i)
\end{align*}
\]

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) andJoinTokens 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 \(t_j = \text{andSplitToken}(t, o_j)\) 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 \(\text{andSplitToken}\) and \(\text{andJoinToken}\) 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 foresee 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 ∈ inArc(node) Enabled(in)
CtlOp(node) = let [in₁, ..., inₙ] = inArc(node)
        let [t₁, ..., tₙ] = firingToken(inArc(node))
        ConsumeAll(((tⱼ, inⱼ)) | 1 ≤ j ≤ n)
        Produce(andJoinToken({t₁, ..., tₙ}), out)
DataOp(node) = forall i ∈ assignments(out) Assign(toᵢ, fromᵢ)

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 GateConditions that evaluate to true (data-based case) or upon GateEvents 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 selectProduce(node), which is constrained to select at each invocation a non-empty subset of arcs outgoing node that satisfy the OrSplitCondition. If there is no such set, the rule cannot be fired.

Constraints for selectProduce
selectProduce(node) ≠ ∅
selectProduce(node) ⊆ {out ∈ outArc(node) | OrSplitCond(out)}¹⁵

This leads to the following instantiation of the WorkflowTransition(node) scheme for or-split gateway nodes. 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 = selectProduce(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) | o ∈ O})
DataOp(node, O) = forall o ∈ O forall i ∈ assignments(o) Assign(toᵢ, fromᵢ)

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 selectConsume: 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 conventional 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 selectConsume 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 selectConsume. As a result selectConsume(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 nodes. 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.

\[
\text{OrJoinGateTransition}(node) = \begin{align*}
\text{let } I &= \text{selectConsume}(node) \text{ in WorkflowTransition}(node, I) \\
\text{where } \\
\text{CtlCond}(node, I) &= (I \neq \emptyset \text{ and } \forall j \in I \text{ Enabled}(j)) \\
\text{CtlOp}(node, I) &= \begin{cases} 
\text{PRODUCE}(\text{orJoinToken}(\text{firingToken}(I)), \text{out}) \\
\text{CONSUMEALL}(\{(t_j, m_j) \mid 1 \leq j \leq n\}) 
\end{cases} \\
\text{where } \\
[t_1, \ldots, t_n] &= \text{firingToken}(I) \\
[m_1, \ldots, m_n] &= I \\
\text{DataOp}(node) &= \forall i \in \text{assignments(out)} \text{ Assign}(to_i, from_i)
\end{align*}
\]

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 selectConsume 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 com-
plex. 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 for-
bidden by the standard to act only as a Merge. Thus those two gateway uses are an instance of
OrJoinGateTransition where selectConsume 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 up-
  stream’ [15, p.80].

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 selectProduce 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 firingTokens 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.

\[
\text{GateTransitionPattern}(\text{node}) =
\begin{align*}
\text{let } I &= \text{selectConsume}(\text{node}) \\
\text{let } O &= \text{selectProduce}(\text{node}) \\
\text{WorkflowTransition}(\text{node}, I, O)
\end{align*}
\]

where

\[
\begin{align*}
\text{CtlCond}(\text{node}, I) &= (I \neq \emptyset \text{ and } \forall \text{ in } I \text{ Enabled}(\text{in})) \\
\text{CtlOp}(\text{node}, I, O) &= \\
\text{PRODUCEALL}(\{(\text{patternToken}(I, o), o) | o \in O\}) \\
\text{CONSUMEALL}(\{t_j, i_n | 1 \leq j \leq n\}) \text{ where} \\
[t_1, \ldots, t_n] &= \text{firingToken}(I) \\
[i_1, \ldots, i_n] &= I
\end{align*}
\]

\[
\text{DataOp}(\text{node}, O) = \forall o \in O \text{ forall } i \in \text{assignments}(o) \text{ Assign}(t_o, f_{o_i})
\]

From this GateTransitionPattern(node) machine one can define the machines above for the various simple gateway nodes. For AND-joins selectConsume chooses all incoming arcs, whereas for OR-joins it chooses exactly (exclusive case) or at least one (inclusive cases). Similarly selectProduce 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.

\[
\text{Event} = \text{StartEvent} \cup \text{IntermEvent} \cup \text{EndEvent}
\]

5.1 Start Events

A start event has no incoming arc (‘no Sequence flow can connect to a Start Event’). 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 triggers 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 ∈ 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\(^\)\(^{20}\)

\[
\text{StartEventTransition}(\text{node}) = \text{choose } e \in \text{trigger}(\text{node}) \text{ StartEventTransition}(\text{node}, e)
\]

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 outgoing 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.

\[
\text{StartEventTransition}(\text{node}, e) = \text{if Triggered}(e) \text{ then Produce(startToken}(e), \text{out}) \text{ ConsumEvent}(e)
\]

**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:

\[
\text{for all } e' \in \text{trigger}(\text{node}) \text{ if Triggered}(e') \text{ then ConsumEvent}(e')
\]

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 selectEvent 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):

\[
\text{if } type(e) = \text{Link then}
\]
\[
\quad Triggered(e) = \text{Enabled}(link)
\]
\[
\quad \text{ConsumEvent}(link) = \text{Consume}(linkToken(link), link)
\]

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:

\[
\text{MultipleStartEventTransition}(node) =
\]
\[
\quad \text{choose } E \subseteq CorrelatedEvent(node)
\]
\[
\quad \text{MultipleStartEventTransition}(node, E)
\]
\[
\text{MultipleStartEventTransition}(node, E) =
\]
\[
\quad \text{if } \forall e \in E \text{ Triggered}(e) \text{ then}
\]
\[
\quad \quad \text{Produce}(startToken(e), out)
\]
\[
\quad \quad \forall e \in E \text{ ConsumEvent}(e)
\]

**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 \texttt{CtlCond} is satisfied, here if the incoming arc is \texttt{Enabled}; as \texttt{CtlOp} it will \texttt{Consume} the firing token. BPMN forsees for end events also a possible \texttt{EVENTOperation}, namely to \texttt{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 \( \text{res}(\text{node}) \) to denote the result defined at a given node.

\[
\text{EVENTTransition}(\text{node}) = \\
\text{if} \ \text{Enabled}(\text{in}) \ \text{then} \\
\text{Consume}(\text{firingToken}(\text{in}), \text{in}) \\
\text{EmitResult}(\text{firingToken}(\text{in}), \text{res}(\text{node}), \text{node})
\]

The type of result and its effect are defined in [15, Table 9.6]. We formalize this by a submachine \texttt{EmitResult}. It \texttt{Sends} messages for results of type \texttt{Message}, where \texttt{Send} denotes an abstract message sending mechanism (which assumes the receiver information to be retrievable from the message). In case of \texttt{Error}, \texttt{Cancel} or Compensation type, via \texttt{EmitResult} an intermediate event is \texttt{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 \texttt{node} and the type of result, by \texttt{targetInternEv}([result, node]).\textsuperscript{22} The node to which \texttt{targetInternEv} belongs is denoted by \texttt{targetInternEvNode}([res, node]). In the \texttt{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 \texttt{CALLBACK} to \texttt{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 \texttt{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 \texttt{Link}: an end event node of type \texttt{Link} is the \texttt{source(link)} of the interpretation of \texttt{link} as a special sequence flow arc, where by the rule \texttt{WFTransition(source(link))} the \texttt{linkTokens}, needed to make the link \texttt{Enabled}, are \texttt{Produced}. As we will see below this may also happen at the source link of a paired intermediate event node of type \texttt{Link}. These tokens will then be consumed by the rule \texttt{WFTransition(target(link))} at \texttt{target(link)}, e.g. a connected start event node of type \texttt{Link} whose incoming arc has been \texttt{Enabled}. We use the same technique to describe that, in case the result type is None and \texttt{node} is a subprocess end node, “the flow goes back to its Parent Process”: we \texttt{Produce} appropriate tokens on the \texttt{targetArc}([node]), which is supposed to lead back to the node where to return in the \texttt{parent(p)} process.

For a result of type \texttt{Terminate} we use a submachine \texttt{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 \texttt{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 \texttt{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 \texttt{TokenSet(p)}.

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

\textsuperscript{22} In case of Error this intermediate event is supposed to be within what is called the \textit{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.
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

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

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$(token(a, t))$
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 $\textit{BoundaryEvents}$. They do not have any incoming arc (“MUST NOT be target for Sequence Flow”), but typically have one outgoing arc denoted again by $\textit{out}$ (“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 outgoing arc, b) exactly one incoming arc if it is of type None, Error or Compensation and at most one incoming 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 $\textit{trigger}(\textit{node})$ to indicate the set of types of (possibly multiple) event triggers that may be associated to $\textit{node}$. 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 $\textit{BoundaryEvents}$.

If two or more event triggers are $\textit{Triggered}$ 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.)

\[
\textsc{IntermEventTransition}(\textit{node}) = \\
\textbf{choose}\ e \in \textit{trigger}(\textit{node}) \ \textsc{IntermEventTransition}(\textit{node}, e)
\]

It remains therefore to define $\textsc{IntermEventTransition}(\textit{node}, e)$ for each type of event $e$ and depending on whether $e$ is a $\textit{BoundaryEv}(e)$ or not.

In each case, the rule checks that the event is $\textit{Triggered}$. The definition of $\textit{Triggered}(e)$ 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 $\textit{Cancel}$ is by definition in [15, Sect.9.3.4 Table 9.8] a $\textit{BoundaryEvent}$ of a transaction subprocess and $\textit{Triggered}$ by an end event of type $\textit{Cancel}$ or a CALLBACK message received during the execution of the transaction. Similarly an intermediate event of type Error or Compensation can be $\textit{Triggered}$ in particular as the result of an end event of corresponding type, see the definition of $\textit{EMITRESULT}$ for end events. The $\textit{EVENTOp}(\textit{node})$ will $\textit{CONSUMEVENT}(e)$, which is defined as for start events adding for the three event types Error, Cancel and Compensation appropriate clauses (typically the update $\textit{Triggered}(e) := \text{false}$).

In the Normal Flow Case where $\textit{BoundaryEv}(e)$ is false, the rule guard contains also the $\textit{CtlCond}$ that the incoming arc—where the activity was waiting for the intermediate event to happen—is $\textit{Enabled}$. Correspondingly there is a $\textit{CtLOP}(\textit{node})$ to $\textit{CONSUM}(\textit{in})$. Where the sequence flow will continue depends on the type of event.

In case of an intermediate event of type $\textit{Link}$, the considered node is the source link node of a paired intermediate event and as such has to $\textit{PRODUCE(linkToken(link), link)}$, 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 $\textit{type}(e) = \textit{None}$ is meant to simply “indicate some state of change in the process”, so that the CtLOP will also $\textit{PRODUCE}$ an appropriate number and type of tokens on the outgoing 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 $\textit{THROWN}$, which means that the corresponding next enclosing $\textit{BoundaryEvent}$

\[23\] Except source link intermediate events, which therefore receive a special treatment in rule $\textsc{IntermEventTransition}(\textit{node}, e)$ below.
occurrence (which we denote by a function $\text{targetIntermEv}$ similar to the one used already in $\text{EMITRESULT}$ above) is $\text{Triggered}$ to handle (catch or forward) the exception, error (corresponding to the $\text{ErrorCode}$ if any) or compensation. In addition the information on the token that triggered the event is stored in the $\text{targetIntermEv}$ by inserting it into a set $\text{excType}$, which is used when the boundary intermediate event is triggered.

In the Exception Case where $\text{BoundaryEv}(e)$ is true, if the activity to whose boundary the intermediate event is attached is $\text{active}$,\textsuperscript{24} the sequence flow is requested to “change the Normal Flow into an Exception Flow” and to $\text{TRYTOCATCH}$ the exception respectively perform the compensation. If there is no match for the exception, it is rethrown to the next enclosing corresponding intermediate $\text{BoundaryEvent}$. If the match succeeds, the $\text{out}$ arc (which we interpret 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 $\text{CTLOP}(\text{node})$ action consists in making this arc $\text{Enabled}$ by an operation $\text{PRODUCE(out)}$.

Every intermediate event of type Compensation attached to the boundary of an activity is assumed by BPMN to catch the compensation (read: to satisfy $\text{ExcMatch}$) 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\textsuperscript{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.

\begin{verbatim}
INTERM_EVENT_TRANSITION \( \text{node}, e \) =
   if $\text{Triggered}(e)$ then
      if not $\text{BoundaryEv}(e)$ then
         if $\text{Enabled}(in)$ then let \( t = \text{firingToken}(in) \)
            $\text{CONSUMEVENT}(e)$
            $\text{CONSUM}(t, in)$
            if $\text{type}(e) = \text{Link}$ then $\text{PRODUCE}(\text{linkToken}(\text{link}),\text{link})$
            if $\text{type}(e) = \text{None}$ then $\text{PRODUCE}(t, out)$
            if $\text{type}(e) = \text{Message}$ then
               if $\text{NormalFlowCont}(\text{msg}(\text{node}),\text{process}(t))$
                  then $\text{PRODUCE}(t, out)$
               else $\text{THROW}(\text{exc}(\text{msg}(\text{node})),\text{targetIntermEv}(\text{node}))$
         if $\text{type}(e) = \text{Timer}$ then $\text{PRODUCE}(\text{timerToken}(t), out)$
         if $\text{type}(e) \in \{\text{Error},\text{Compensation},\text{Rule}\}$ then $\text{THROW}(e,\text{targetIntermEv}(e))$
      if $\text{BoundaryEv}(e)$ then
\end{verbatim}

\textsuperscript{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]

\textsuperscript{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).

21
The Petri net model for tasks in

Both tasks and subprocesses can contain iterative components of different loopType usually will take its execution time without this time being furthermore analyzed in the workflow viewed as a unit process and not “defined as a flow of other activities” (ibid. p. 53), though it may and executed” (ibid.), so that atomicity refers to the fact that as part of a business process the task is given business process “an end-user and/or an application are used to perform the Task when it is atomicity. Typically the action underlying the given task is intended to represent that within the atomicity of tasks does not imply their zero-time execution.

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 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 true for each Triggered event of type Timer, Message or Rule.

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 INTMEVENTTRANSITIION(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 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 true for each Triggered event of type Timer, Message or Rule.

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:

\[
\text{Activity} = \text{Task} \cup \text{SubProcess} \cup \text{IterProc}
\]

\[
\text{IterProc} = \text{Loop} \cup \text{MultiInstance} \cup \text{AdHoc}
\]

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.
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 MultiInstance 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, 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 incoming and/or outgoing 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 incoming and/or outgoing 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 Executing 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 ReadyForExecution, which in turn depends on the particular type of the task, as does the task EXECUTION. The standard considers eight types for tasks:

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

A task of type Service or User is defined to be ReadyForExecution 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(inMsg(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 performers 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 incoming arc and the associated process has no start event, or task has an incoming 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)) = true.

Tasks of type Send, Manual or Script are designed to unconditionally Execute the associated action, namely to SEND(mssg(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 selectInputSets 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) ∈ 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.

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

29 The standard document leaves it open whether the service executing agent respectively the human performers 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 outgoing 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 Task 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.

\[
\text{TaskTransition}(\text{task}) = \begin{cases} 
\text{if Enabled}(\text{in}) \text{ then} \\
\text{if ReadyForExec}(\text{task}) \text{ then let } t = \text{firingToken}(\text{in}) \\
\text{let } i = \text{selectInputSets}(\text{SomeAvail}(\text{inputSets}(\text{task}))) \\
\text{EXEC(task, inputs}(i)) \\
\text{currInput}(\text{task}) := i \\
\text{seq} \\
\text{if Completed(task, t) then} \\
\text{[PRODUCEOUTPUT(outputSets}(\text{task}), \text{currInput}(\text{task}))] \\
\text{[PRODUCE(taskToken}(\text{task}, t), \text{out})]} \\
\text{where} \\
\text{PRODUCEOUTPUT(outputSets}(t), i) = \\
\text{choose } o \in \text{outputSets}(t) \text{ with Defined}(\text{outputs}(o)) \text{ and IORules}(t)(o, i) = \text{true} \\
\text{Emit}(\text{outputs}(o)) \\
\end{cases}
\]

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

\[
\text{EXEC}(t, i) = \\
\begin{cases} 
\text{SEND}(\text{inMssg}(t)) & \text{if type}(t) \in \{\text{Service, User}\} \\
\text{RECEIVE}(\text{mssg}(t)) & \text{if type}(t) = \text{Receive} \\
\text{SEND}(\text{mssg}(t)) & \text{if type}(t) \in \{\text{Send}\} \\
\text{CALL}(\text{performer}(\text{action}(t, i)), \text{action}(t, i)) & \text{if type}(t) \in \{\text{Script, Manual}\} \\
\text{EXEC(taskRef}(t), i) & \text{if type}(t) = \text{Reference} \\
\text{skip} & \text{if type}(t) = \text{None} \\
\end{cases}
\]

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 incoming and (at most) one outgoing arc.

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

- **before** the to be iterated activity 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 nodes 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.

\[
\text{LoopTransition}(node) = \begin{cases} \text{if Enabled}(in) \text{ then} \\
\text{let } t = \text{firingToken}(in) \\
\text{LoopEntry}(node, t) \\
\text{seq} \\
\text{if } \text{testTime}(node) = \text{before} \text{ then} \\
\text{while } \text{loopCond}(node, t) \text{ LoopBody}(node, t) \\
\text{if } \text{testTime}(node) = \text{after} \text{ then} \\
\text{until } \text{loopCond}(node, t) \text{ LoopBody}(node, t) \\
\text{[seq LoopExit}(node, t)] \\
\end{cases}
\]

where

\[
\text{LoopBody}(n, t) = \\
\text{loopCounter}(node, t) := \text{loopCounter}(node, t) + 1 \\
\text{iterBody}(node, \text{loopToken}(t, \text{loopCounter}(node, t) + 1)\text{[inputs(currInput(node))])}
\]

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 answered whether this is considered as completion of the loop or not (see the example for “improper looping” in Fig.10.51 p.129).

\[
\text{LoopEntry}(n, t) = \\
\text{loopCounter}(n, t) := 0 \\
\text{[Consume}(t, in)\text{]} \\
\text{[currInput}(n) := \text{selectInputSets}(\text{SomeAvail}(\text{inputSets}(n)))\text{]}
\]

\[
\text{LoopExit}(n, t) = \\
\text{if Completed}(n, t) \text{ then} \\
\text{[ProduceOutput}(\text{outputSets}(n), \text{currInput}(n))\text{]} \\
\text{[Produce}(\text{loopExitToken}(t, \text{loopCounter}(n, t)), \text{out})\text{]} \\
\text{Completed}(n, t) = \text{LoopCompleted}(n, t) \text{ if } n \in \text{Loop}(t)
\]

**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 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 ≤ miNumber the predicate Completed(iterBody(node, miToken(t, i)[. . .])) 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 ≤ miNumber the predicate Completed(iterBody(node, miToken(t, i)[. . .])) 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))])
  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)[. . .]))
COMPL_MULT_INST_EXIT has to keep track of whether the initially empty set of those activity instances, which have \textit{AlreadyCompleted} and have passed their continuation tokens to the outgoing arc, has reached the prescribed number \textit{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 \textit{NewCompleted} we once more \textsc{Produce} a token on the outgoing arc \textit{out}—if the \textit{TokenTime(complexMiFlowCond)} condition allows us to do so, in which case we also insert the instance into the set \textit{AlreadyCompleted}. Since the context apparently is distributed and since the standard document contains no constraint on \textit{TokenTime(complexMiFlowCond)}, at each moment more than one instance may show up in \textit{NewCompleted}.\footnote{The description of the case \textit{miFlowCond} = \textit{One} in the standard document is ambiguous: the wording \textit{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.} Therefore we use a selection function \textit{select_{NewCompleted}} to choose an element from the set \textit{NewCompleted}\footnote{In ASM terminology this is a derived set, since its definition is fixed and given in terms of other dynamic locations, here \textit{Completed} and \textit{AlreadyCompleted}.} of multiple instances that have \textit{Completed} but not yet produced their continuation token.\footnote{If one prefers not to describe any selection mechanism here, one could instead use the \textsc{forall} construct as done in the \textit{else} branch. This creates however the problem that it would not be impossible for more than \textit{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 \textsc{Produce} a token on the outgoing arc.} In the following definition \textit{n} is supposed to be a multiple instance activity node with parallel \textit{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.

\begin{verbatim}
COMPL_MULT_INST_EXIT(n, t) = // for miOrdering(n) = Parallel
    alreadyCompleted := \emptyset // initially no instance is completed
    seq
        while alreadyCompleted \neq \{ i | i \leq m\text{iNumber}(n) \} do
            if NewCompleted(n, t) \neq \emptyset then
                if \textit{tokenNo(complexMiFlowCond)} \text{<} \textit{tokenNo(complexMiFlowCond)}
                    then
                        if \textit{TokenTime(complexMiFlowCond)} then
                            let i_0 = select_{NewCompleted} in
                            \textsc{Produce}(\textit{miExitToken}, i_0, out)
                            \textsc{Insert}(i_0, alreadyCompleted)
                            \textsc{ProduceOutput}(outputSets(n), currInput(n))
                        else forall i \in NewCompleted(n, t) \textsc{Insert}(i, alreadyCompleted)
                where
                    NewCompleted(n, t) = \{ i \leq m\text{iNumber}(n) | \textit{Completed(iterBody}(n, \textit{miToken}(t, i)[\ldots])) \text{ and } i \notin alreadyCompleted \}
                The \textsc{EveryMultInstExit} machine is an instance of \textsc{ComplMultInstExit} where \textit{tokenNo} is the number (read: cardinality of the set) of all to-be-considered activity instances and the \textit{TokenTime} is any time.

\textsc{EveryMultInstExit}(n, t) = \textsc{ComplMultInstExit}(n, t)
    where
        \textit{tokenNo(complexMiFlowCond)} = \{ i | i \leq m\text{iNumber}(n) \} |
        \textit{TokenTime(complexMiFlowCond)} = true
\end{verbatim}
OneMultInstExit is an instance of ComplMultInstExit where tokenNo = 1 and the TokenTime is any time.

OneMultInstExit(n, t) = ComplMultInstExit(n, t) where

\[ \text{tokenNo}(\text{complexMiFlowCond}) = 1 \]
\[ \text{TokenTime}(\text{complexMiFlowCond}) = \text{true} \]

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 adHocOrdering function is provided to specify either a parallel execution (the default case) or a sequential one.

Notably the definition of when an adhoc activity is Completed is left to a monitored predicate AdHocCompletionCondition, which “cannot be defined beforehand” (ibid.p.132) and is required to be “determined by the performers of the activities”. Therefore the execution of the rule for an adhoc process continues as long as the AdHocCompletionCondition 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 AdHocCompletionCondition 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]
\[ \text{let } t = \text{firingToken}(\text{in}) \]
\[ \text{[Consume}(t, \text{in})] \]
\[ \text{[let } i = \text{selectInputSets}(\text{SomeAvail}(\text{inputSets}(\text{node}))) \]
\[ \text{currInput(node) := } i \]
\[ \text{while not AdHocCompletionCond(node, t)} \]
\[ \text{if adHocOrder(node) = Parallel then } \forall a \in \text{innerAct}(node) \text{ do } a[\text{inputs}(i)] \]
\[ \text{if adHocOrder(node) = Sequential then } \forall a_0, \ldots, a_n : \text{innerAct}(node) \]
\[ \text{foreach } j < n \text{ do } a_j[\text{inputs}(i)] \]
\[ \text{seq LoopExit(node, t)} \]
\[ \text{where Completed(node, t) = AdHocCompletionCond(node, t)} \]

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 $A \subseteq_{\text{multi}} B$ to denote that $A$ is a multi-set of elements from $B$.

\[
\text{UnconstrainedAdHocTransition}(\text{node}) = \begin{cases} 
\text{if Enabled}(\text{in}) \text{ then} & \\
\text{let } t = \text{firingToken}(\text{in}) & \\
\quad \text{[Consume}(t, \text{in})] & \\
\quad \text{let } i = \text{selectInputSets}(\text{SomeAvail}(\text{inputSets}(\text{node}))) & \\
\quad \text{currInput}(\text{node}) := i & \\
\quad \text{while not AdHocCompletionCond}(\text{node}, t) & \\
\quad \text{choose } A \subseteq_{\text{multi}} \text{innerAct}(\text{node}) & \\
\quad \text{forall } a \in A \text{ do } a[\text{inputs}(i)] & \\
\quad \text{seq LOOPExit}(\text{node}, t) & \\
\quad \text{where Completed}(\text{node}, t) = \text{AdHocCompletionCond}(\text{node}, t) & 
\end{cases}
\]

Many issues remain open with such an interpretation. For example, can an activity within an ad hoc 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 hoc 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 incoming and (at most) one outgoing 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 conveys the token information related to this link to 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 committee 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 feasible, 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 selectProduce and selectConsume).
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).\textsuperscript{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

\texttt{WorkflowTransitionInterpreter =}
\texttt{let node = selectNode\{n | n \in Node and Enabled(n)\}}
\texttt{let rule = selectWorkflowTransition\{r | r \in WorkflowTransition and Fireable(r, node)\}}
\texttt{rule}

The behavioral rule scheme (form of rules in \texttt{WorkflowTransition}):

\texttt{WorkflowTransition(node) =}
\texttt{if EventCond(node) and CtlCond(node)}
\texttt{and DataCond(node) and ResourceCond(node) then}
\texttt{DataOP(node)}
\texttt{CtlOP(node)}
\texttt{EVENTOP(node)}
\texttt{RESOURCEOP(node)}

9.2 Gateway Rules

\texttt{AndSplitGateTransition(node) = WorkflowTransition(node)}
\texttt{where}
\texttt{CtlCond(node) = Enabled(in)}
\texttt{CtlOP(node) =}
\texttt{let t = firingToken(in)}
\texttt{CONSUME(t, in)}
\texttt{PRODUCEALL\{\{andSplitToken(t, o), o \in outArc(node)\}\}}
\texttt{DATAOP(node) = //performed for each selected gate}
\texttt{forall o \in outArc(node) forall i \in assignments(o) ASSIGN(to_i, from_i)}

\texttt{AndJoinGateTransition(node) = WorkflowTransition(node)}
\texttt{where}
\texttt{CtlCond(node) = forall in \in inArc(node) Enabled(in)}
\texttt{CtlOP(node) =}
\texttt{let [in_1, \ldots, in_n] = inArc(node)}
\texttt{let [t_1, \ldots, t_n] = firingToken(inArc(node))}
\texttt{CONSUMEALL\{\{t_j, in_j\} | 1 \leq j \leq n\}}
\texttt{PRODUCE\{andJoinToken\{t_1, \ldots, t_n\}, out\}}
\texttt{DATAOP(node) = forall i \in assignments(out) ASSIGN(to_i, from_i)}

\textsuperscript{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].
**9.3 Event Rules**

**StartEventTransition**(node) =

\[\text{choose } e \in \text{trigger}(node) \rightarrow \text{StartEventTransition}(node, e)\]

**StartEventTransition**(node, e) =

\[\text{if } \text{Triggered}(e) \text{ then } \text{PRODUCE}(\text{startToken}(e), \text{out}) \]  
\[\text{CONSUMEVENT}(e)\]

**EndEventTransition**(node) =

\[\text{if } \text{Enabled}(in) \text{ then } \text{CONSUME}(\text{firingToken}(in), \text{in}) \]  
\[\text{EMITRESULT}(\text{firingToken}(in), \text{res}(node), \text{node})\]

**GateTransitionPattern**(node) =

\[\text{let } I = \text{selectConsume}(node) \text{ in } \text{WORKFLOWTRANSITION}(node, I)\]

where

\[\text{CtlCond}(node, I) = (I \neq \emptyset \text{ and } \forall j \in I \text{ Enabled}(j))\]

\[\text{CTLOP}(node, I) = \]  
\[\text{PRODUCE}((\text{patternToken}(\text{firingToken}(I)), o) \mid o \in O)\]  
\[\text{CONSUME}((t_j, \text{in}) \mid 1 \leq j \leq n)\]  
\[\text{where}\]
\[\{t_1, \ldots, t_n\} = \text{firingToken}(I)\]
\[\{\text{in}_1, \ldots, \text{in}_n\} = I\]

\[\text{DATAOP}(node, O) = \forall o \in O \forall i \in \text{assignments}(o) \text{ ASSIGN}(\text{to}_i, \text{from}_i)\]
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)
        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
      if active(targetAct(e)) then
        ConsumeEvent(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)
        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

\text{\textsc{TaskTransition}}(task) = \begin{cases} \text{if} \ Enabled(\text{in}) \ \text{then} \\
\text{if} \ \text{ReadyForExec}(task) \ \text{then} \ \text{let} \ t = \text{firingToken}(\text{in}) \\
[\text{CONSUME}(t, \text{in})] \\
\text{let} \ i = \text{selectInputSets}(\text{SomeAvail}(\text{inputSets}(task))) \\
\text{EXEC}(task, \text{inputs}(i)) \\
\text{currInput}(task) := i \\
[\text{seq}] \\
\text{if} \ \text{Completed}(task, t) \ \text{then} \\
[\text{PRODUCEOUTPUT}(\text{outputSets}(task), \text{currInput}(task))] \\
[\text{PRODUCE}(\text{taskToken}(task, t), \text{out})] \\
\text{where} \\
\text{PRODUCEOUTPUT}(\text{outputSets}(t), i) = \\
\text{choose} \ o \in \text{outputSets}(t) \ \text{with} \ \text{Defined}(\text{outputs}(o)) \ \text{and} \ \text{IORules}(t)(o, i) = \text{true} \\
\text{EMIT}(\text{outputs}(o)) \\
\end{cases}

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

\text{\text{EXEC}}(t, i) = \begin{cases} \text{SEND}(\text{inMssg}(t)) \quad \text{if} \ type(t) \in \{\text{Service, User}\} \\
\text{RECEIVE}(mssg(t)) \quad \text{if} \ type(t) = \text{Receive} \\
\text{SEND}(mssg(t)) \quad \text{if} \ type(t) \in \{\text{Send}\} \\
\text{CALL}(\text{performer}(\text{action}(t, i)), \text{action}(t, i)) \quad \text{if} \ type(t) \in \{\text{Script, Manual}\} \\
\text{EXEC}(\text{taskRef}(t), i) \quad \text{if} \ type(t) = \text{Reference} \\
\text{skip} \quad \text{if} \ type(t) = \text{None} \\
\end{cases}

\text{\text{LoopTransition}}(\text{node}) = \begin{cases} \text{if} \ Enabled(\text{in}) \ \text{then} \\
\text{\text{LOOPENTRY}}(\text{node}, t) \\
\text{seq} \\
\text{if} \ \text{testTime}(\text{node}) = \text{before} \ \text{then} \\
\text{while} \ \text{loopCond}(\text{node}, t) \ \text{LOOPBODY}(\text{node}, t) \\
\text{if} \ \text{testTime}(\text{node}) = \text{after} \ \text{then} \\
\text{until} \ \text{loopCond}(\text{node}, t) \ \text{LOOPBODY}(\text{node}, t) \\
[\text{seq} \ \text{LOOPEXIT}(\text{node}, t)] \\
\text{where} \\
\text{LOOPBODY}(\text{node}, t) = \\
\text{loopCounter}(\text{node}, t) := \text{loopCounter}(\text{node}, t) + 1 \\
\text{iterBody}(\text{node}, \text{loopToken}(t, \text{loopCounter}(\text{node}, t) + 1), \text{inputs}(\text{currInput}(\text{node}))) \\
\text{\text{LOOPENTRY}}(\text{node}, t) = \\
\text{loopCounter}(\text{node}, t) := 0 \\
[\text{CONSUME}(t, \text{in})] \\
[\text{currInput}(\text{node}) := \text{selectInputSets}(\text{SomeAvail}(\text{inputSets}(\text{node})))] \\
\text{\text{LOOPEXIT}}(\text{node}, t) = \\
\text{if} \ \text{Completed}(\text{node}, t) \ \text{then} \\
[\text{PRODUCEOUTPUT}(\text{outputSets}(\text{node}), \text{currInput}(\text{node}))] \\
[\text{PRODUCE}(\text{loopExitToken}(t, \text{loopCounter}(\text{node}, t)), \text{out})] \\
\text{Completed}(\text{node}, t) = \text{LoopCompleted}(\text{node}, t) \ \text{if} \ \text{n} \in \text{Loop}(\text{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 i₀ = selectNewCompleted in
          PRODUCE(miExitToken(t, i₀), out)
          INSERT(i₀, 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 = selectInputSets(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)
\[
\text{ADHOC\textsc{Transition}}(\text{node}) = \begin{cases} \text{if Enabled(in) then} \\
\text{let } t = \text{firingToken(in)} \\
\text{[CONSUME}(t, \text{in})] \\
\text{let } i = \text{selectInputSets} (\text{SomeAvail(inputSets(node)))} \\
\text{currInput(node) := } i \\
\text{while not AdHocCompletionCond(node, t) do} \\
\text{if adHocOrder(node) = Parallel then } \forall a \in \text{innerAct(node) do } a[\text{inputs}(i)] \\
\text{if adHocOrder(node) = Sequential then } \text{let } <a_0, \ldots, a_n> = \text{innerAct(node)} \\
\text{foreach } j < n \text{ do } a_j[\text{inputs}(i)] \\
\text{seq LOOPEXIT(node, t) } \text{where } \text{Completed(node, t) = AdHocCompletionCond(node, t)}
\end{cases}
\]

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

![Fig. 1. Viewing FSM instructions as control state ASM rules](image)

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, \(\text{cond}_\nu\) (for \(1 \leq \nu \leq n\)) represents the input condition \(\text{in} = a_\nu\) (reading input \(a_\nu\)) and \(\text{rule}_\nu\), the output action \(\text{out} := b_\nu\) (yielding output \(b_\nu\)), which goes together with the \(\text{ctl\_state}\) update to \(j_\nu\). 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 \(\text{ctl\_state}\) that assumes values in a not furthermore structured finite set
- two input and output locations \(\text{in}, \text{out}\) 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 \(\text{ctl\_state}\) and of the \(\text{out}\) location, is turned into an ASM step consisting of the simultaneous update of a set of locations, namely via multiple assignments of the form \(\text{loc}(x_1, \ldots, x_n) := \text{val}\), 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 \(\text{condition}\) is extended to arbitrary ASM-state expressions, namely formulae in the signature of the ASM states. They are called \(\text{guards}\) 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 \(\text{synchronous parallelism}\), which yields a clear concept of \(\text{locally described global state change}\), 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 \(\text{ctl\_state} := i\) into the guard and \(\text{ctl\_state} := j\) into the multiple assignments of the rules, we obtain the definition of a \(\text{basic ASM}\) as a set of instructions of the following form, called ASM \(\text{rules}\) to stress the distinction between the parallel execution model for basic ASMs and the sequential single-instruction-execution model for traditional programs:

\[
\text{if } \text{cond then Updates}
\]

where \(\text{Updates}\) stands for a set of function updates \(f(t_1,\ldots,t_n) := t\) 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) \(\text{multi-agent runs}\) turns FSMs into globally asynchronous, locally synchronous Codesign-FSMS \([27]\) and similarly basic ASMs into \(\text{asynchronous ASMs}\) (see \([12, \text{Ch.6.1}]\) 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 \(\text{Set}\), which satisfy a certain (possibly runtime) \(\text{Property}\):

\[
\text{forall } x \in \text{Set} \[\text{with Property}(x)\] \text{ do rule}(x)
\]

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

Where the sequential execution of first \(M\) followed by \(N\) is needed we denote it by \(M \text{ seq } N\), 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:

\[
\text{foreach } i \leq n \text{ do rule}(i) = \text{ rule}(1) \text{ seq rule}(2) \text{ seq } \ldots \text{ seq rule}(n)
\]

\(^{38}\) For the special role of \(\text{in}/\text{output}\) locations see below the classification of locations.

\(^{39}\) More precisely: to execute one step of an ASM in a given state \(S\) (s.t. \(\text{cond}\) is true in \(S\)), compute all expressions \(t_i, t\) in \(S\) occurring 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](image-url)
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 \[12, \text{Ch.2.2.3}\] 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.

\[
\text{choose } x \in \text{Set} \text{[with Property}(x)\text{][do ]}
\]

\[
\text{rule } (x)
\]

This stands for the ASM executing rule(\( x \)) for some element \( x \), which is arbitrarily chosen among those which are element of \( \text{Set} \) and satisfy the selection criterion Property. Sometimes we omit the key word do. The parts \( \in \text{Set} \) and with 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 zip((\( x_i \), (\( y_i \)), 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 selection function, to select an element from the underlying set of Candidates, and writes instead of choose \( c \in \text{Cand do } R(c) \) as follows, where \( R \) is any ASM rule:

\[
\text{let } c = \text{select(Cand) in } R(c)
\]

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.


References


On defining the behavior of OR-joins in business process models

Egon Börger\textsuperscript{1} *, Ove Sørensen\textsuperscript{2}, and Bernhard Thalheim\textsuperscript{2}

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

Abstract. The recent literature on business process modeling notations contains numerous contributions to the so-called OR-join (or inclusive merge gateway) problem. We analyze the problem and present an approach to solve it without compromising any of the two majors concerns that are involved: a) a clear semantical definition (design), which also clarifies what has to be implemented to achieve the intended generality of the construct, and b) a comprehensive set of static and dynamic analysis methods (verification of properties of business process models using the construct). We provide a conceptually simple scheme for dynamic OR-join synchronization policies, which can be implemented with low run-time overhead and allow the practitioner to effectively link the design of business process models with OR-joins to an analysis of the intended model properties. The definitions have been experimentally validated by a graph-based simulator.\textsuperscript{3}

1 Introduction

A major problem for reliable software-based system development is to guarantee that the system does what it is supposed to do. This holds also for computer-assisted enterprise information and management systems, where IT technologists (system designers, software engineers and programmers) have to understand and realize the system behaviour that is expected by business process experts. A technical, but crucial instance of this general problem concerns the concept of OR-join. This concept is present in various workflow and business process modeling languages and seems to be used with different understandings in different commercial workflow systems or even worse, for some workflow languages, differently by users of the language and by the implementation. Our goal in this paper is to clarify the issues involved and to contribute to solving the problem by an accurate definition that is easy to understand, can be experimentally validated, is not biased by the underlying framework used for the definition, puts the various approaches in the literature into a clear perspective and provides a rigorous basis for implementing various verifiable synchronization policies for business process models with OR-joins.

The OR-join problem has various aspects which have been dealt with in numerous papers. First of all it is a \textit{problem of semantics}, in the sense that in some languages the behavioral meaning of the OR-join is not defined in a precise enough way to exclude undesired ambiguity. Two examples of such languages are the language of event process chains (EPCs) \cite{1} (see for example the analysis in \cite{26,19}) and the current BPMN standard \cite{8} (see for example the analysis in \cite{9,12}). Furthermore, even if the semantics of the OR-join is mathematically well-defined, this definition may be regarded as too complicated to support practitioners in their design work where they need a reliable understanding of the expected behavior of the business process models they are defining; see for example the view expressed in \cite{15} for the fixpoint-semantics-based definition of the semantics of EPCs in \cite{17,18}.

The OR-join problem appears in the literature also as a \textit{verification method problem} in the sense that even where a behavioral definition is given, the computational cost of mechanically verifying

\textsuperscript{*} The work of the first author is supported by a Research Award from the Alexander von Humboldt Foundation (\textit{Humboldt Forschungspreis}), hosted by the Chair for Information Systems Engineering of the third author at the Computer Science Department of the University of Kiel/Germany.

\textsuperscript{3} The simulator has been developed by the second author and is part of his Diplom Thesis \cite{24}.
some desired properties of models based upon that definition of OR-joins may be deemed to be too expensive, so that restrictions are imposed on the allowed process models. See for example the OR-join treatment in the YAWL language [27], which for reasons of complexity does not consider nested OR-joins. In [32], which seems to be a reenactment of [31], two “problems with OR-join semantics as defined in [27]” (quote of the title of [32, Sect.2.2]) are identified and the restriction is eliminated, based upon the work in [30] (see also [33]). However, the general solution in [30] comes with a computational complexity that is considered as too high by the authors of [10] and [12, Sect.4.5.3] and motivated the proposal there of a less expensive algorithm for a more restricted interpretation of OR-joins.

Complexity concerns seem to have motivated also the proposal in [13,14,15] to use an experimentally justified recursive set of rules for defining a comprehensive class of ‘structured’ models without OR-join problems.

What one can observe here are efforts to trade the generality of a semantically well-defined OR-join concept for the complexity of checking properties of models containing such OR-joins, notably the enabledness property for OR-joins. The perspective of such a dichotomy may also have influenced the fact that many commercial workflow tools simply impose syntactic restrictions on the OR-join, as came out of the study [22].

However, from the conceptual point of view the situation is not as bad as it appears from the discussion in the literature, which is largely influenced by a bias towards some conventional but unnecessarily restrictive ways of defining and verifying workflow features, in particular Petri nets and various ad hoc extensions. Concerning verification it should be remembered that as in traditional engineering disciplines, also in software engineering verification is not limited to mechanical (whether static or runtime) property checks. Professional reasoning to provide quality assurance in an engineering discipline typically exploits the full range of available rigorous scientific methods, which goes from well-founded testing of characteristic patterns through traditional mathematical reasoning to interactive computer-assisted or—in the limit case—even fully automated proofs or exhaustive model checking. This holds also for correctness considerations for business process models, which ultimately need to be deeply rooted in the application domain knowledge one can hardly expect to be ever completely automated and analyzable by static analysis tools (see the notion of ground model in [4]).

Concerning definition methods, it should be remarked that the need for application domain based reasoning goes together with the need NOT to restrict the range of descriptive means by an a priori imposed formal language, as too often has happened in computer science theory and seems to happen again in the workflow and business process modeling domain (as a recent example see the fight for Petri nets versus Pi-calculus [25], a representative for many such detrimental battles that happened in the so called Formal Methods domain of computer science). If one wants to be successful with high-level models, from where code can be generated using sophisticated application-independent compilation techniques, one has to avoid the straitjacket of specific formal languages as long as the main concerns are related to application domain problems and not to their formal (let alone software) representation. This applies in particular to the OR-join construct as it is used in most business process languages.

The main result of this paper is a simple, precise and unbiased definition of the OR-join scheme. It captures the originally intended generality in a direct way and clearly shows the problems this generality brings for the concept itself as well as for its implementations. The definition uses only general, process-related and accurately definable notions every business analyst and system designer understands so that it can serve as a basis for communication when it comes to decide upon appropriate instances of the general definition. The general purpose algorithmic language we use allows one also to use any rigorous method whatsoever to establish specific properties of interest for such instances. In addition we show that by our definition the various OR-join approaches in the literature can be put into a uniform perspective. Since in doing this we will refer to most of the relevant literature, there will be no specific Related Work section in this paper.4

4 Just before submitting the final version of this paper to the editor we found a paper [29] which also proposes to use run-time information for defining the precise behavior of OR-joins. The authors propose
Our definition is based upon a framework developed recently in [7], motivated by the goal to define a complete rigorous semantical model for the current BPMN standard [8] and its forthcoming extension 2.0.\(^5\) We start here from scratch and recapitulate a few of the definitions from [7] that are useful for the discussion of the OR-join construct. To graphically represent our examples we use the BPMN notation without further explanation.

In Sect. 2 we review the intuitive understanding of the OR-join as it appears from related investigations in the literature and explain what is called the OR-join problem. In Sect. 3 we sketch the framework that is used in Sect. 4 to define a precise semantics for the general intuitive understanding of the OR-join. In [24] this definition is extended to the case where multiple tokens may occur in a cyclic diagram.

To introduce the OR-join model, we will use the technique of stepwise ASM refinement [3]. Adopting a token-based view of workflow semantics, we start out with the base case of acyclic workflows where joins can determine their enabledness locally (this includes XOR- and AND-joins). Next, we add OR-joins to the – still cycle-free – model where the non-local information about the state of the entire workflow that the intuitive OR-joins semantics requires is provided by introducing a special type of synchronization token that firing flow objects place in their downstream.

As next refinement step we consider cyclic workflows. They are BPMN standard conform, but their semantics is underspecified. This underspecification is due to the underlying synchronization problem. One has to expand on the BPMN standard if one wants to account for this. As a first step we desynchronize cyclic control flow by associating a token set to each cyclic token. This approach is extended in [24] to the case where multiple tokens may occur within a cycle. Essentially two new types of flow objects are added to describe barrier-like behaviour, used to create cleanly nested structures inside a workflow that synchronize control flow between multiple cycle-iterations and can be proved to be free from deadlocks. For the experimental validation of these extensions and of the definitions in this paper the second author has developed a simulator that can visualize the execution of BPMN workflows [24].

The reader who is acquainted with the problem may go immediately to Sect. 4 and consult Sect. 3 only should the need be felt.

2 Analysis of OR-Join Requirements

In this section we try to review the intuitions behind the OR-join concept. The literature offers a variety of interpretations of the OR-join as a control flow construct where different computation paths are synchronized in a way that depends on runtime conditions and ranges from the XOR (select exclusively exactly one) to the AND-join (synchronize all) behavior.

To start we quote two typical descriptions. The first one is the BPMN standard document description, which uses the naming Inclusive Gateway used as a Merge:

If there are multiple incoming Sequence Flow, one or more of them will be used to continue the flow of the Process. That is, 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. [8, p.81]

The standard document leaves it open how to determine when an incoming sequence flow (read: an arc leading to the OR-join node) is “expecting a signal based on the upstream structure

\(^5\)\text{for this purpose a global history log on all consumed or produced tokens. Our model works with a less expensive and simpler run-time information structure, which is tailored to the synchronization problem of OR-joins. In [29] only rather special cyclic workflow diagrams can be proved to be without deadlocks.}

\(^5\)\text{The attribute business happens to be part of the established nomenclature, although the processes described by the BPMN as well as the OR-join problem are of general nature and not restricted to modeling business applications.}
of the Process”, except for the indication that this is a process instance feature and therefore data-dependent and runtime-defined. Also the notion of upstream structure is not further described (except for calling loops downstream activities, see below).

In [20] one can find an analysis, carried out in terms of EPCs, of some natural definitions for which paths an OR-join should wait for to complete their computation. In the presence of a parenthesis structure, which links the incoming arcs of the OR-join one-to-one to the outgoing arcs of a preceding OR-split, it appears to be natural to require the OR-join to synchronize the threads on all and only those paths that have been activated at that OR-split (see the Occam-like OR-join semantics in Sect. 4.2.). If there is no such underlying syntactical graph structure, one could ask the OR-join node to take a special action for one completing thread (e.g. the first one if there is any) and then wait for the others to complete6, or to react upon each path completion (en bloc for multiple simultaneous completions or choosing among them one after the other, as happens in the Multi-Merge pattern interpretation in [22]), or in the limit case to behave as the AND-join.

A similar (possibly intended) specification hole is found in the description of the workflow pattern analogue of the OR-join in [28], called there synchronizing merge:

A point in the workflow process where multiple paths converge into one single thread. If more than one path is taken, synchronization of the active threads needs to take place. If only one path is taken, the alternative branches should reconverge without synchronization. It is an assumption of this pattern that a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete.

Nothing is said to explain when a “path is taken” or a “branch has been activated” except for the further clarification that there is a notion of a round in which other (all?) branches are expected “to complete” (how? normally? abruptly due to some failure?). For details see the critical analysis in [5].

The common feature of the above two and other descriptions in the literature seems to be that some synchronization is to be performed7 and that this should happen only for currently active threads. The debated question is how to determine whether a thread is (potentially?) active. In [32, Sect.2.1] the attempt is made to answer this question on the basis of the following more detailed definition of what there is called the informal semantics of an OR-join.

An OR-join task is enabled at a marking iff at least one of its input conditions is marked and it is not possible to reach a marking that still marks all currently marked input conditions (possibly with fewer tokens) and at least one that is currently unmarked. If it is possible to place tokens in the unmarked input conditions of an OR-join in the markings reachable from the current marking, then the OR-join task should not be enabled and wait until either more input conditions are marked or until it is no longer possible to mark more input conditions.

This description reveals what some authors call the non-local nature of the OR-join semantics. More accurately one should speak about the non-local character of means needed to determine, for this interpretation of the construct, whether or not an OR-join is enabled: it does not suffice to check for tokens in its incoming (or somehow nearby) arcs, as done to establish enabledness of transitions in Petri net and coloured Petri net workflow descriptions, but one has to evaluate some global markings, namely all those reachable from the current marking, in order to check whether some of them enable an additional incoming arc of the OR-join without disabling any of the ones already enabled in the current marking. Since it can turn out to be difficult to implement efficient algorithms for such an evaluation, some workflow systems and some authors prefer to restrict the semantics of OR-joins in order to obtain simple means of checking the enabledness condition.

6 This is a form of the so-called Discriminator pattern in [28].
7 Therefore the naming OR-join is rather misleading: synchronization has much more to do with AND (logical conjunction) than with OR (logical disjunction). The fact that a certain runtime variation is involved in establishing which threads are to be synchronized is closer to choice and non-determinism than to disjunction. In this sense synchronizing merge or simply synchronization are more appropriate names.
We advocate to separate the two different concerns involved. We first provide in Sect. 4 a simple
precise definition of the desired intuitive meaning of the OR-join, without making any restrictive
assumptions and without inventing for the purpose yet another workflow language [27,35,33] (how
often a new one [21]?) or extension of Petri nets [32,33,34]. Our definition reflects the global features
of the intended synchronization in a direct way, avoiding the well-known problems, discussed for
example in [12], one has with Petri net based formulations of the semantics of business process
models. These problems are due to the local nature of what a Petri net transition can do and
have motivated various extensions of Petri nets and related verification techniques to cope with
OR-join and cancellation features in business process models, see for example [35,33,34]. Only
after a clear definition one should use whatever scientific or mathematical means are available to
decide upon and to analyze instances of the general definition and to establish or check properties
of models with OR-joins. Obviously this includes, but is not restricted to, mechanical checks of
the enabledness condition by existing tool sets. We believe that the need to solve the challenging
correctness problem when modeling business processes makes it compulsory to have an easy to
understand definition of the OR-join behavior and its properties, even more if it is felt that tricky
and difficult algorithms to compute such properties are unavoidable.

3 The Modeling Framework

In this section we borrow from [7] that part of the business process modeling framework that
allows one to capture the intuition of the OR-join by a concise and clear definition. For the sake of
definiteness we use for the discussion of workflow constructs a BPMN-based terminology, without
making conceptually or methodologically restricted assumptions so that our results can be applied
to other business process model notations as well.

3.1 Abstract State Machines

We use for our descriptions Abstract State Machines(ASMs), an extension of Finite State Machines
by a concept of most general state and of synchronous parallelism for state transformations. Per
step an arbitrary number of simultaneous updates is allowed, which are described by finitely many
rules that at each 'step' are executed simultaneously (synchronous parallelism). The form of the
rules is as follows:

\[
\begin{align*}
\text{if } \text{cond} & \text{ then Updates} \\
\text{where Updates} & \text{ stands for a set of function updates } f(t_1, \ldots, f_n) := t \text{ built from expressions } t_i, t \\
& \text{ and an } n\text{-ary function symbol } f. \text{ Equivalently one can use the graphical or textual FSM notation}
\end{align*}
\]

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 \( f \). Equivalently one can use the graphical or textual FSM notation
depicted in Fig. 1, where \( i, j_1, \ldots, j_n \) are internal (control) states as known from FSMs.

Fig. 1. View (control state) ASM rules as generalized FSM instructions

Since the mathematical definition of the semantics of ASMs supports their intuitive understanding
as pseudo-code working over abstract data types, we abstain from repeating the definition here
and refer the interested reader for this to the AsmBook [6]. To define the various interpretations of the OR-join as different instantiations of one abstract model we make use of the ASM refinement method defined in [3].

3.2 Business Process Diagrams

As common in the field, we mathematically represent any business process as a graph. The nodes represent the workflow objects, where activities are performed depending on a) resources being available, b) data or control conditions to be true and c) events to happen, as described by transition rules associated to nodes. These rules define the meaning of the corresponding workflow constructs. The arcs define the graph traversal, i.e. the order in which the workflow objects are visited for the execution of the associated rules.

We freely use the usual graph-theoretic concepts, for example source(arc), target(arc) for source and target node of an arc, pred(node) for the 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 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.

All the workflow transition rules, associated to nodes to describe the meaning of the workflow construct associated to this node, take the following form (usually instantiated by additional parameters). They state upon which events and under which further conditions on the control flow, the underlying data and the availability of resources, the rule can fire to perform specific operations on the underlying data (‘how to change the internal state’) and control (‘where to proceed’), to possibly trigger new events (besides consuming the triggering ones) and to operate on the resource space to take possession of the needed (or to release not any more needed) resources.

\[
\text{WorkflowTransition}(\text{node}) = \\
\text{if } \text{EventCond}(\text{node}) \text{ and CtlCond}(\text{node}) \text{ and DataCond}(\text{node}) \text{ and ResourceCond}(\text{node}) \text{ then } \\
\text{DataOp}(\text{node}) \\
\text{CtlOp}(\text{node}) \\
\text{EventOp}(\text{node}) \\
\text{ResourceOp}(\text{node})
\]

A workflow or business process modeling language interpreter is a set of such rules, covering all language constructs, together with a scheduler to choose at each moment a node where a rule can be fired, which is the case when its guard is true in the current state. In this way one can define for example the semantics of the BPMN standard by an interpreter with rules (more precisely rule schemes) for each BPMN flow object (activities, events, gateways) [7]. For the discussion of the OR-join problem we can focus the discussion on gateways only (see Sect. 3.4). Furthermore, for this discussion events and resources play no role and therefore will not be mentioned any more.

3.3 Token-Based Sequence Flow Interpretation

Although the BPMN standard document declares to use the token-based interpretation of control flow only for illustrative purposes [8, p.35], for the sake of definiteness we represent it mathematically by associating tokens—elements of a set Token—to arcs, using a dynamic function token(arc). A token typically includes information on (the processID of) the process instance to which it belongs. Typically token(arc) denotes a multiset of tokens currently residing on arc.

\[\text{token} : \text{Arc} \rightarrow \text{Multiset}(\text{Token})\]

\[\text{We deliberately avoid introducing yet another category of graph items, like the so-called places in Petri nets, whose only role would be to hold these tokens.}\]
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. This condition is typically required to be an atomic quantity formula stating that the number of tokens currently associated to \( \mathit{in} \) (read: the cardinality of \( \mathit{token}(\mathit{in}) \), denoted \( |\mathit{token}(\mathit{in})| \)) is at least the input quantity \( \mathit{inQty}(\mathit{in}) \) required at this arc.

\[
\mathit{Enabled}(\mathit{in}) = (|\mathit{token}(\mathit{in})| \geq \mathit{inQty}(\mathit{in}))
\]

Correspondingly the control operation \( \mathit{CtLOp} \) of a workflow usually consists of two parts, one describing how many tokens are \( \mathit{Consumed} \) on which incoming arcs and one describing which tokens are \( \mathit{Produced} \) on which outgoing arcs in a quantity as indicated by a function \( \mathit{outQty}(\mathit{out}) \). We use macros to describe consuming resp. producing tokens on a given arc and then generalize them to produce or consume all elements of a given set. We also define the most frequent case where tokens are simply \( \mathit{Passed} \) from an incoming to an outgoing arc. \( \mathit{outQty}(\mathit{out}) \) denotes the number of tokens one wants to be produced on arc \( \mathit{out} \). In many applications \( \mathit{inQty}(\mathit{in}), \mathit{outQty}(\mathit{out}) \) are assumed to take the default value 1.

\[
\mathit{Consume}(t, \mathit{in}) = \mathit{Delete}(t, \mathit{inQty}(\mathit{in}), \mathit{token}(\mathit{in}))
\]
\[
\mathit{Produce}(t, \mathit{out}) = \mathit{Insert}(t, \mathit{outQty}(\mathit{out}), \mathit{token}(\mathit{out}))
\]

\[
\mathit{Pass}(t, \mathit{in}, \mathit{out}) =
\begin{align*}
&\mathit{Delete}(t, \mathit{inQty}(\mathit{in}), \mathit{token}(\mathit{in})) \\
&\mathit{Insert}(t, \mathit{outQty}(\mathit{out}), \mathit{token}(\mathit{out}))
\end{align*}
\]

The macro is easily generalized to sets of pairs of tokens and arcs:

\[
\mathit{ConsumeAll}(X) = \mathit{forall } x \in X \mathit{Consume}(x)
\]
\[
\mathit{ProduceAll}(Y) = \mathit{forall } y \in Y \mathit{Produce}(y)
\]

**Remark** This use of macros allows one to easily adapt the abstract token model to its extensions, like the ones we use in Sect. 4, and to different instantiations by a concrete token model. For example, if a token is simply defined as a pair \((\mathit{proc}(t), \mathit{pos}(t))\) of the process instance it belongs to and the arc where it is positioned, then it suffices to refine the macro for \( \mathit{Pass} \)ing a token \( t \) from \( \mathit{in} \) to \( \mathit{out} \) by updating the second token component, namely from its current position value \( \mathit{in} \) to its new value \( \mathit{out} \):

\[
\mathit{Pass}(\mathit{in}, \mathit{out}, t) = (\mathit{pos}(t) := \mathit{out})
\]

The use of abstract \( \mathit{Delete} \) and \( \mathit{Insert} \) operations instead of directly updating \( \mathit{token}(a) \) 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 \( \mathit{Delete}(\mathit{in}, t) \) and \( \mathit{Insert}(\mathit{out}, t) \) one may have \( \mathit{in} = \mathit{out} \).

### 3.4 Gateway Nodes

Gateways are used to describe the splitting (divergence) or merging (convergence) of control flow in the sense that tokens can ‘be merged together on input and/or split apart on output’ [8, p.68]. Both splitting and merging come usually in two forms, which are related to the propositional operators \textbf{and} and \textbf{or}, namely a) to create parallel or synchronize multiple actions and b) to select (one or more) among some alternative actions. For the sake of a clear separation of the different merge/split features and without loss of generality, 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 control flow. It is easy to show that each BPMN process can be transformed into a semantically equivalent BPMN Best Practice Normal Form.

- **BPMN Best Practice Normal Form.** [8, p.69] Only gateways have multiple incoming or multiple outgoing arcs and furthermore they never have both multiple incoming and multiple outgoing arcs.
For the sake of illustration we formulate and explain now the two AND gateway node rule specializations of the general WorkflowTransition rule scheme, to prepare the reader for the discussion of the OR-join gateway rule in the next section. Since the focus of the OR-join analysis is on token-based control, we skip here and for the AND-join below the formulation of the not control related conditions and operations, like the DATAOp(node), which in BPMN is an ASSIGNOperation performed at each outgoing arc.

To fire an AND-split node requires—besides the node-specific conditions on data, events and resources—that Enabled holds for its unique incoming arc in. Upon firing, the rule in particular CONSUMes the prescribed number of tokens and PRODUCes on each of the finitely many outgoing arcs (elements of outArc(node)) the prescribed number of tokens. These outgoing tokens are typically viewed as triggering parallel subprocesses, which may be required to be synchronized later within the process where they have been generated. For this reason, tokens produced at split gateways are often assumed to carry some information about the origin and maybe also about their brothers and sisters with whose descendants they may have to be synchronized at a later stage. This is the case in BPMN where tokens serve the purpose of "dividing of the Token for parallel processing within a single Process instance" [8, p.35]. We describe this by an abstract function andSplitToken whose values may depend on the incoming token and the outgoing arc. We will use this function below for the discussion of the OR-join gateway rule, where for the sake of definiteness we represent the function concretely as follows, concatenating the incoming token with the chosen arc to record the information about the path the token went through at this split node:

\[\text{andSplitToken}(t, o) = t.o\]

We also take here the view of BPMN where the prescribed quantity for consuming or producing tokens on incoming respectively outgoing arcs of AND and OR gateways is 1. To express that upon firing the AndSplitGateTransition one has to select on the unique incoming arc in one of its token(in) to be Consumed we use a function firingToken({in}). For later reference we use this function as defined on non-empty subsets of inArc(node).

\[
\text{AndSplitGateTransition}(node) = \text{WorkflowTransition}(node)
\]

where

\[
\begin{align*}
\text{CtlCond}(node) &= \text{Enabled}(in) \\
\text{CtlOp}(node) &= \\
&\quad \text{let } t = \text{firingToken}({in}) \\
&\quad \text{CONSUMe}(t, in) \\
&\quad \text{PRODUCEAll}({(\text{andSplitToken}(t, o), o) | o \in \text{outArc}(node)})
\end{align*}
\]

Frequently splitting a computation into finitely many branches comes with a later join of these branches (or even more branches that may be due to further intermediate splits). To fire an AND-join node requires—besides the node-specific conditions on data, events and resources—that Enabled holds for each of its finitely many incoming arcs in \(\in\) inArc(node). Upon firing, the gateway CONSUMes the prescribed number (here 1) of the tokens on every incoming arc and PRODUCes on its unique outgoing arc out the prescribed number (here 1) of tokens. Since a join node typically has a synchronization purpose, the relation between the incoming token and the outgoing token often reflects this feature. We formulate this dependence by a function andSplitToken whose values depend on the incoming tokens. The function firingToken chooses here a set of tokens, containing one token from token(in) for each incoming arc in.

\[
\text{AndJoinGateTransition}(node) = \text{WorkflowTransition}(node)
\]

where

\[
\begin{align*}
\text{CtlCond}(node) &= \forall \text{ in } \in \text{inArc}(node) \text{ Enabled}(in) \\
\text{CtlOp}(node) &= \\
&\quad \text{let } \{in_1, \ldots, in_n\} = \text{inArc}(node) \\
&\quad \text{let } \{t_1, \ldots, t_n\} = \text{firingToken}(\text{inArc}(node))
\end{align*}
\]
ConsumeAll(\{(t_i, \text{in}) \mid 1 \leq i \leq n\})
Produce(\text{andJoinToken(}\{t_1, \ldots, t_n\}), \text{out})

The reader will have noticed that we did not specify the firing tokens by \text{let } t_i = \text{firingToken(in}_i),\text{ because this would mean that one can select the tokens on the incoming arcs independently from each other. Instead the function firingToken typically will select “matching tokens” with respect to a to be defined matching condition.}

4 OR-Join Definition

In this section we use the framework explained in Sect. 3 to define a precise semantics for the general supposedly intuitive understanding of the OR-join. We first define in Sect. 4.1 the OR-split gateway rule along the lines of the AND-split gateway rule, but adding a mechanism to describe how to choose among alternative subsets of outgoing arcs (instead of selecting the entire set outArc(node)). We then adapt this selection mechanism to describe the synchronization features of the OR-join rule. To separate two different concerns related to the OR-join problem we split the discussion into two parts, one for acyclic graphs (Sect. 4.2) and one for graphs with cycles (Sect. 4.3).

4.1 OR-Split Gateway Rule

An OR-split is similar to the AND-split, but instead of producing tokens on every outgoing arc, this may happen only on a non-empty subset of them. The chosen alternative depends on certain conditions OrSplitCond(o) to be satisfied that are associated to outgoing arcs o. For example in the BPMN standard, OrSplitCond(o) is an associated GateCond(o) or a GateEvent(o). We reflect this choice among the various alternatives by an abstract function selectProduce(node), which is constrained to select at each invocation a non-empty subset of arcs outgoing node that satisfy the OrSplitCondition. The BPMN standard document for example imposes default gates to guarantee for a valid process that every call of this function yields a non empty set. A special version of this interpretation of OR-split nodes is to additionally require that with each selection a singleton set (exclusive choice) is determined, whether based upon an event or a data condition, e.g. by trying the alternatives out in an a priori fixed manner (in BPMN called data-based or event-based XOR-split). However, by the nature of their role these selection functions often are not static (compile-time definable), but dynamic functions, whose values depend on the runtime state. We will exploit this in the next section for the description of the OR-join behavior.

Constraints for selectProduce

selectProduce(node) \neq \emptyset
selectProduce(node) \subseteq \{out \in outArc(node) \mid OrSplitCond(out)\}\)

This leads to the following instantiation of the WorkflowTransition(node) scheme for OR-split gateway nodes. 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 in the AND-split rule we use a function, here orSplitToken, to express the type of tokens to be produced on outgoing arcs. in denotes the unique incoming arc.

\text{OrSplitGateTransition(node) = WorkflowTransition(node)}
\text{where}
\text{let } \{\text{in}\} = \text{inArc(node)}
\text{let } O = \text{selectProduce(node)} \text{ in}

\text{orSplitToken} = \text{orSplitToken(in, O)}

\text{OrSplitCond(out)}

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) and EventCond(node, O) in the guard of OrSplitGateTransition.
\[\text{CtlCond}(\text{node}) = \text{Enabled}(\text{in})\]
\[\text{CtlOp}(\text{node}, O) = \]
\[
\text{let } t = \text{firingToken}\{\text{in}\}\]
\[
\text{CONSUME}(t, \text{in})
\]
\[
\text{PRODUCERALL}\{\text{orSplitToken}(t, o), o \in O\}\}
\]

Since \text{ANDSplitGateTransition} is an instance of \text{ORSplitGateTransition}, namely with the selection function required to yield the entire set \(\text{outArc}(\text{node})\), we speak in the following only of split nodes when we mean an AND split or OR split gateway at a node; similarly for join nodes with correspondingly specialized synchronization condition.

### 4.2 OR-Join for Cycle-Free Models

In this section the graphs are assumed to be acyclic. For simplicity of exposition but without loss of generality we add here and in the next section to the BPMN Best Practice Normal Form assumption the \textbf{Unique Start Node Assumption} that each graph has exactly one start node.\(^{10}\)

Thus every node in the graph is connected to the start node by a path.

Sometimes it is claimed that “the non-locality of OR-joins can even raise problems to the effect that it is \textit{impossible} to define a formal semantics \ldots that is fully compliant with the informal semantics” \cite[p.6]{15}, but as the authors of \cite{10} point out, the problem is not in the \textit{definition} of what they call the OR-join \textit{firing rule}, but in a) the definition of when this rule should be considered as \textit{enabled} and b) in finding efficient algorithms to compute this enabledness property.

In fact, to describe the OR-join gate transition rule it suffices to adapt to a function \textit{selectConsume} the mechanism used above to describe via \textit{selectProduce} the (decisions taken about the) possible alternatives when firing an OR-split transition rule.

We explicitly separate the two distinct features one has to consider for the constraints to impose on such a \textit{selectConsume} function: the enabledness condition for each selected arc and the synchronization condition that the selected arcs are exactly the ones to synchronize. We represent the undisputed conventional token constraint as part of the control condition in the \text{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 is the synchronization constraint for \textit{selectConsume} functions. Before investigating it 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 \textit{selectConsume}.

Thus \textit{selectConsume}(\text{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 \text{node}, with the usual effect.

\[
\text{ORJoinGateTransition}(\text{node}) = \text{WorkflowTransition}(\text{node})
\]
\[
\text{where}
\]�
\[
\text{let } I = \text{selectConsume}(\text{node}) \text{ in}
\]
\[
\text{CtlCond}(\text{node}, I) = (I \neq \emptyset \text{ and for all } i \in I \text{ Enabled}(i))
\]
\[
\text{CtlOP}(\text{node}, I) =
\]
\[
\text{PRODUCER}(\text{orJoinToken}(\text{firingToken}(I)), \text{out})
\]
\[
\text{CONSUMEALL}\{\text{1} \leq i \leq n\} \text{ where}
\]
\[
\{t_1, \ldots, t_n\} = \text{firingToken}(I)
\]
\[
\{i_1, \ldots, i_n\} = I
\]

The \textit{selectConsume} function in the \text{ORJoinGateTransition} serves to express on which arcs one has to wait for tokens of the indicated type from the to-be-synchronized threads \cite{28}, in terms of the BPMN standard document on which arcs we “are expecting a signal based on the upstream structure of the Process” \cite[p.81]{8}. The real question is first of all which synchronization condition

\(^{10}\)To a graph with multiple nodes that can be used for starting a sequence flow, one can add a split gateway that splits to the multiple start nodes from a new unique starting node. This can be an AND-split or an OR-split, depending on the interpretation of the use of multiple start nodes. In BPMN it is disjunctive for start events and conjunctive for implicit start nodes.
one wants to impose as constraint on the \( \text{select}_\text{Consume} \) function,\(^{11}\) and then which means we have to compute values of the function once it is defined (read: the enabledness condition for OR-join rule instances).

It is surprising to see that the workflow and business process oriented literature on the theme deals with this issue without ever referring to well known and sophisticated techniques to handle synchronization problems in distributed computing. This may be another theme where “business process modelers can learn from programmers” \(^{15}\).

We try in the following to investigate some variations of the \texttt{ORJOINGATETRANSITION} rule proposed in the literature to put them into a unified perspective. We hope that by doing this the sometimes hidden assumptions or motivations of those proposals become clear and can be evaluated for an informed decision on the intended OR-join synchronization behavior.

**“Informal semantics” of OR-join** We start with an analysis of the proposal quoted in Sect. 2 for what is called the informal semantics of the OR-join. The literature contains some sophisticated algorithms to compute the OR-join enabledness property for this interpretation of the OR-join, see for example \([10]\) which improves on \([31]\). It comes down to determine (why restricted to static analysis means?) all computation paths that may lead to enabling additional arcs entering this node. One can specify this requirement in an accurate way by providing some additional (in an optimized version not really expensive to produce) runtime information on what is of concern, namely for which potential synchronization requests a join gateway node may still have to handle the synchronization.

Since by the unique start node assumption we know that synchronization requests are produced only at split gateway nodes, we can capture the requirement for the “informal” OR-join semantics in our model by “informing” all synchronization points, which are reachable from a split node, as soon as possible about tokens that may have to be synchronized at the join node and to keep this information up to date during subsequent decision points. The latter may exclude some of the—up to this decision point possible—paths for a token. This comes up to send an advance notice, for each token created at a split node, to all reachable join nodes and to maintain this information up to date until the token arrives at the synchronization point or takes a path from where that point cannot be reached. This can be described in the model by the following refinement:

- add in the split and join node transition rules synchronization analogues to the token production and consumption submachines in \texttt{CTLOP},
- add in the join node rules the intended synchronization counterpart \texttt{CtlCondSync}(node, I) to the \texttt{CtlCond}(node, I), checking whether for each synchronization token an enabling token is present.

Here are the details of this refinement step.

**Split gate transition refinement** Let \( \text{node} \) be a split node and \( \text{out} \) any arc outgoing \( \text{node} \) where a token \( t \) enabling the unique incoming arc in \texttt{PRODUCES} a token \( t.\text{out} \). This starts a new computation path at \( \text{out} \) that may need to be synchronized with other computation paths started simultaneously at this \( \text{node} \) (or with some final segment of some computation paths started upstream, i.e. at nodes from where \( \text{node} \) can be reached\(^{12}\)). We place an additional synchronizer copy of \( t.\text{out} \) on each reachable arc that enters a join node, more precisely for each path that starts with \( \text{out} \) and leads to an arc \( a \) entering a join node, we place a synchronizer copy of \( t.\text{out} \) on \( a \). We denote the set of these join arcs by \( \text{AllJoinArc}(\text{out}) \) and record the synchronizer token copy placed there in a location \( \text{syncToken}(\text{arc}) \). This allows us

\footnote{Although this question is in no way related to the meaning of OR as expressing some alternatives for firing the join rule, we keep the name \text{select}_\text{Consume}, instead of (for example) \text{synthesize}, to show that different interpretations of this function correspond to different choices made for the synchronization discipline at OR-joins.}

\footnote{This complication is needed as long as it is allowed to synchronize computation paths started at different split nodes, as for example in the BPMN standard \([8]\). It is avoided for example in Occam-like OR-join interpretations discussed below.}
to define analogues `PRODUCE(ALL)SYNC` of `PRODUCE(ALL)` to handle the placement of synchronizer tokens. By calling a corresponding submachine `CONSUMESYNCALL` we also delete the synchronization copy of the fired `t` for each `o ∈ outArc(node)` from each `i ∈ AllJoinArc(o)`. This reflects that once `t` is fired, the request for its potential synchronization is replaced by a request for potential synchronization of the children token `PRODUCED` by (firing the rule triggered by) `t`, and only those. The refined rule is formulated below in Sect. 4.3.

- **Join gate transition refinement** Let `node` be a join node. `CtlOp` is refined as for split nodes by adding the `CONSUMESYNCALL` and `PRODUCESYNCALL` submachines, called upon appropriate sets of tokens to a) consume the synchronization tokens that, once the to-be-synchronized tokens have been fired, have served their purpose, and to b) produce new synchronization tokens for the tokens the join produces. In addition we refine the `CtlCond(node, I)` by adding the intended synchronization condition `CtlCondSync`. In the case of the informal OR-join semantics we are formalizing here, `CtlCondSync` expresses that `I` is a synchronization family at `node`, which means a set of incoming arcs with non-empty `syncToken` sets such that all other incoming arcs (i.e. those not in `I`) have empty `syncToken` set (read: are arcs where no token is still announced for synchronization so that no token will arrive any more (from upstream) to enable such an arc).

The definition of the macros `PRODUCE`, `CONSUME` and their extensions to sets can be copied for synchronization tokens by replacing `token` with `syncToken`.\(^ {13} \) The quantity functions `inQty`, `outQty` are skipped because by assumption at split or join rules, on each involved arc only 1 token is consumed or produced.

\[
\begin{align*}
&\text{PRODUCESYNC}(t, in) = \text{INSERT}(t, \text{syncToken}(in)) \\
&\text{CONSUMESYNC}(t, in) = \text{DELETE}(t, \text{syncToken}(in)) \\
&\text{PRODUCESYNCALL}(Y) = \text{forall } y \in Y \text{ PRODUCESYNC}(y) \\
&\text{CONSUMESYNCALL}(X) = \text{forall } x \in X \text{ CONSUMESYNC}(x)
\end{align*}
\]

For **split gate transition rules** the `CtlOp(node)` submachine is refined by adding the following two submachines. We use the instance of the functions `orSplitToken` and `andSplitToken` explained already above, namely the trace notation `t.out`, to record the start at `out` of a computation path triggered by `t`, a path which is (potentially) to be synchronized with other computation paths started at the same node (or upstream) so that the same `t.out` is placed into `syncToken`.

\[
\begin{align*}
&\text{PRODUCESYNCALL}\{(t, o, i) \mid i \in AllJoinArc(o), o \in O\} \\
&\text{CONSUMESYNCALL}\{(t, i) \mid i \in AllJoinArc(o) \text{ forsome } o \in outArc(node)\}
\end{align*}
\]

For **join gate transition rules** the `CtlOp(node)` submachine is refined by a refinement of the function `firingToken(I)` and by adding the following two submachines.

\[
\begin{align*}
&\text{PRODUCESYNCALL}\{\{(joinToken(t_1, \ldots, t_n), in) \mid in \in AllJoinArc(out)\} \\
&\text{CONSUMESYNCALL}\{\{t_i, in\mid in \in AllJoinArc(out), 1 \leq i \leq n\} \cup \{\{t_i, in\mid 1 \leq i \leq n\}\}
\end{align*}
\]

`firingToken(I)` is refined to select among the enabling and synchronization tokens on arcs in `I` a maximal common token prefix `t` such that the following condition holds:

\[
\text{forall } 1 \leq i \leq n \ t_i = t.\text{rest} \in \text{token}(\text{in}i) \cap \text{syncToken}(\text{in}i).
\]

Correspondingly we refine `orJoinToken` resp. `andJoinToken` to `joinToken(t_1, \ldots, t_n) = t`.

The synchronization counterpart `CtlCondSync(node, I)` added as conjunct to `CtlCond(node, I)` expresses that all the selected arcs are involved in a potentially forthcoming synchronization, but no other incoming arc.

\[
\begin{align*}
&\text{CtlCondSync(node, I)} = \\
&\text{forall } i \in I \text{ syncToken}(i) \neq \emptyset \text{ and forall } i \in \text{inArc(node)} \setminus I \text{ syncToken}(i) = \emptyset
\end{align*}
\]

\(^ {13} \) The reader who knows the ASM refinement method [3] knows that one could avoid this repetition by parameterizing the macros by a function `tok`, which can then be instantiated to `token` or `syncToken`. Similarly for an instantiation of `FireForAll(rule, Z)` for `rule = PRODUCE, CONSUME, etc.}`
Illustration by Example  Fig. 2 illustrates the preceding definition. Here are some typical cases.

Case 1: at split 1 only one token is produced, a token entering A2. Then the arcs entering join 1, join 3, join 4, join 5 on the path from A1 to End 1 and only those receive synchronization tokens, so that the rule at these join nodes can fire immediately when the token coming from A2 arrives, since no further synchronization has to take place.

Case 2: at split 1 only two tokens are produced, one entering A1 and one entering A2. Subcase 2.i (i=1,2): at split 2 only one token is produced, namely to enter Bi. Then synchronization tokens are produced on the path from A2 to End 1 as in case 1. The additional synchronization tokens produced at join 3, join 4, join 5 on the two paths from A1 to End 1 have three effects. They prevent the rule at join 3 from firing until in case 2.1 the decision to produce a token to enter activity B1 (and not B2) has been taken (whereby the synchronization token is deleted from the arc connecting B2 with join 3 as well as from the arcs entering join 4 and join 5 on the path from B2 to End 1), or in case 2.2 until the token produced at the exit from B2 has arrived to be synchronized with the token coming from A2. At join 5 two potential synchronizations are required when a token leaves split 1 to enter A1, one on the arc exiting D1 and one on the arc exiting join 4. The first of these two synchronization requests holds in case 2.1 until the token produced upon exiting split 2 to enter B1 arrives at join 5, in case 2.2 until the decision to produce a token to enter activity B2 (and not B1) has been taken. Symmetrically for the second synchronization request. At join 4 still no synchronization is necessary since the synchronization tokens produced there between exiting split 1 and entering join 3 are deleted upon entering join 3 and by assumption no (synchronization) token is produced on paths going through A3 or A4.

The other possible cases are analogous.

Remark on cancellation. To include the consideration of cancellation regions [30,33] in a business process diagram it suffices to update, in addition to a cancellation action that takes place at a node, syncToken at all synchronization points that are downstream a node in the cancellation region of node.

Variations of OR-join semantics  Neither the literature nor the BPMN standard clarify satisfactorily what are the required properties for the OR-join semantics. This implies that there is no binding contract against which one could verify the correctness of a rigorous definition for the semantics of the OR-join. It also implies that it is not clear how to define that a concrete BPMN
diagram is actually well-specified. Instead, there are some variations of the OR-join semantics we are going to shortly characterize here.

In the above description of the “informal semantics” for the OR-join, every potential synchronization token is dismissed from $\text{syncToken}(\text{node})$ whenever a runtime choice made in a transition upstream $\text{node}$ excludes a path. Therefore $\text{CtlCondSync}(\text{node}, \text{selectConsume}(\text{node}))$ becomes true only when all these decisions have been taken. One could replace this cautious approach by a definition of an eager synchronization model, where at a join node only synchronization requests from the next preceding split node are taken, as for example in a situation where nested synchronizations are not needed. Our model can easily be adapted to this case, namely by refining the $\text{AllJoinArc}$ function to a function $\text{NextJoinArc}(o)$ that yields the set of all next arcs downstream $o$ that enter a join node. In the same way one can treat other forms of “scope controlled” synchronization schemes, e.g. the Occam-like interpretation sketched below.

In a similar way one can adapt our model by refining $\text{CtlCondSync}(\text{node}, I)$ to describe synchronization schemes with timeout conditions or similar runtime features.

The very special interpretation of OR-joins by the Synchronizing Merge pattern \[28\] needs neither synchronization tokens nor a $\text{CtlCondSync}$, since every token on any single incoming arc is enough to fire the rule. To describe this as an instance of the $\text{OrJoinGateTransition}(\text{node})$ it suffices to refine $\text{selectConsume}(\text{node})$ to yield singleton sets.

**Occam-like OR-join semantics** An OR-join semantics in the style of the parallel programming language Occam and its Transputer implementation \[11,16\] has for each split $\text{node}$ a well-defined synchronization node $\text{sync}(\text{node})$ where all the processes triggered by a token $t$ at $\text{node}$ are synchronized before one can proceed with the next task after $\text{sync}(\text{node})$, so that in particular $\text{selectConsume}(\text{sync}(\text{node})) = \text{inArc}(\text{sync}(\text{node}))$. This also yields a well-structured discipline for nested synchronizations, which makes the synchronization method explained for acyclic graphs work also in the presence of parallel subprocesses created by parallel processes. Since $\text{sync}(\text{node})$ is known at design time, the production of synchronization tokens is reduced to send from a split $\text{node}$ each produced token $t.o$ to its corresponding synchronization arc $\text{sync}(o)$; the synchronization token consumption is reduced to consume at join nodes these tokens once all to be synchronized processes are ready for their synchronization.

### 4.3 OR-Join for Models with Cycles

In a non-Occam like OR-join semantics one has the problem to define whether and how the synchronization of “upstream” started processes should be combined with the synchronization of “downstream” started processes, e.g. iterations, since such cases are not excluded by the informal and similarly unstructured interpretations of the OR-join semantics. This problem has triggered various research efforts. It is mentioned also in the BPMN standard document, where however no indication about the intended solutions is provided:

Incoming Sequence Flow that have a source that is a downstream activity (that is, is part of a loop) will be treated differently than those that have an upstream source. They will be considered as part of a different set of Sequence Flow from those Sequence Flow that have a source that is an upstream activity. \[8, p.82\]

Thinking of tokens in terms of up-/downstream does not solve the problem of cyclic workflows. According to the definition of “upstream” in the BPMN standard \[8, p.25\], a node is “upstream” regarding some other node, if there is a path in the workflow from the first to the second node. The BPMN standard gives no definition for “downstream”, but seems to implicitly refer to the inverse of “upstream” whenever “downstream” is mentioned. Thus in cyclic workflows, two flow objects can easily be upstream (or downstream) regarding each other in both directions. Therefore this property cannot be used as a discriminator for synchronization. Instead, we will individually group each token that can potentially exhibit cyclic behaviour.

Some further structure is needed to appropriately deal with cyclic workflows.
Token Sets To speak about the synchronization of tokens in cycles needs the ability to express that certain tokens belong together, whereas others do not. To express such a concept we introduce token sets, i.e. sets of tokens which are viewed as a coherent group when a join fires. We will use the token sets to assign new token sets to tokens at paths that have later to be synchronized and to distinguish tokens in cycles by appropriately assigned token sets. In this section we prepare the needed purely syntactical refinement, which is used in the next section to handle the problem of cycles.

We will make sure that each token \( t \) is a member of exactly one token set \( \text{tokenSet}(t) \). We assume that upon a start event a token set \( \text{tokenSet}(t) \) is generated for the start token \( t \). When new tokens \( t' \) appear during the computation, their \( \text{tokenSet}(t') \) has to be defined, as happens in particular in the join rules. In the purely syntactical refinement defined in this section, the new tokens are declared to belong to the same token set as the firing tokens.

We also have to refine the concept of Enabledness to guarantee that each time only tokens of one token set \( ts \) are considered.

\[
\text{Enabled}(\text{in}, ts) = \{ \text{token}(\text{in}) \cap ts | \text{in} \geq \text{inQty}(\text{in}) \}
\]

Similarly we impose on firingToken that each time only tokens belonging to one token set are selected.

\[
\text{if } \text{firingToken}(\text{node}) = \{t_1, \ldots, t_n\} \text{ then forall } 1 \leq i \leq n \text{ tokenSet}(t_i) = \text{tokenSet}(t_1)
\]

This leads to the following refinement of the AND-join rule:

\[
\text{ANDJOINGATETRANSITION}(\text{node}) = \text{WORKFLOWTRANSITION}(\text{node})
\]

where

\[
\begin{align*}
\text{let } & \{in_1, \ldots, in_n\} = \text{inArc}(\text{node}) \\
\text{let } & \{t_1, \ldots, t_n\} = \text{firingToken}(\text{inArc}(\text{node})) \\
\text{let } & ts = \text{tokenSet}(t_1) \\
\text{CtlCond}(\text{node}) = & \text{forall } in \in \text{inArc}(\text{node}) \text{ Enabled}(in, ts) \\
\text{CTLOP}(\text{node}) = & \text{CONSUMEALL}((\{t_i, in_i\} | 1 \leq i \leq n)) \\
& \text{PRODUCE}((\text{andJoinToken}(\{t_1, \ldots, t_n\}), \text{out})) \\
& \text{tokenSetAndJoinToken}((\{t_1, \ldots, t_n\})) := ts \\
& \text{CONSUMESYNCALL}((\{t_i, in\} | in \in \text{AllJoinArc}(\text{out}), 1 \leq i \leq n) \cup \{\{t_i, in_i\} | 1 \leq i \leq n\}) \\
& \text{PRODUCESYNCALL}((\{\text{andJoinToken}(t_1, \ldots, t_n), in\} | in \in \text{AllJoinArc}(\text{out}))
\end{align*}
\]

Synchronization at an OR-join only happens among tokens of the same token set. We therefore refine the synchronization part of the control condition as follows, where \( ts \) is the given token set:

\[
\text{CtlCondSync}(\text{node}, I, ts) = \\
\text{forall } i \in I \text{ syncToken}(i) \cap ts \neq \emptyset \text{ and forall } i \in \text{inArc}(\text{node}) \setminus I \text{ syncToken}(i) \cap ts = \emptyset
\]

With these preparations we can now refine the Or-join to work only on tokens of the token set underlying the to-be-fired tokens on the selected arcs:

\[
\text{ORJOINGATETRANSITION}(\text{node}) = \text{WORKFLOWTRANSITION}(\text{node})
\]

where

\[
\begin{align*}
\text{let } & I = \{in_1, \ldots, in_n\} = \text{selectConsume}(\text{node}) \\
\text{let } & \{t_1, \ldots, t_n\} = \text{firingToken}(I) \\
\text{let } & ts = \text{tokenSet}(t_1) \text{ in} \\
\text{CtlCond}(\text{node}) = & (I \neq \emptyset \text{ and forall } i \in I \text{ Enabled}(i, ts) \text{ and } \text{CtlCondSync}(\text{node}, I, ts)) \\
\text{CTLOP}(\text{node}) = & \text{PRODUCE}((\text{orJoinToken}(\text{firingToken}(I)), \text{out})) \\
& \text{tokenSetAndJoinToken}((\text{orJoinToken}(\text{firingToken}(I)))) := ts \\
& \text{CONSUMEALL}((\{t_i, in_i\} | 1 \leq i \leq n)) \\
& \text{CONSUMESYNCALL}((\{t_i, in\} | in \in \text{AllJoinArc}(\text{out}), 1 \leq i \leq n) \cup \{\{t_i, in_i\} | 1 \leq i \leq n\}) \\
& \text{PRODUCESYNCALL}((\{\text{orJoinToken}(t_1, \ldots, t_n), in\} | in \in \text{AllJoinArc}(\text{out}))
\end{align*}
\]
Obviously this refinement is purely incremental (conservative). Therefore the refined model is backwards compatible with the previous one. We are now ready to assign new token sets to tokens at paths that have later to be synchronized and to distinguish tokens in cycles by appropriately assigned token sets.

**Breaking the Cycles** We use token sets to create tokens that can be distinguished from other tokens in a process instance. This happens at the outgoing arcs of splits that are part of a cycle. We make here the assumption, which is released in the furthermore refined model in [24], that for each cycle and token set, in each path in that cycle there is at each moment at most one token of that token set. Here is the definition of cycle we are using, where the upper index + denotes the transitive closure:

\[
\text{cycle}(a) \iff \text{source}(a) \in \text{succ}^+(\text{target}(a))
\]

We modify the **OrSplitGateTransition** to create on its outgoing cyclic arcs tokens that belong to a new (completely fresh) token set. The new token sets are assumed to be created by a function \(\text{genTokenset}\), so that for each chosen outgoing arc which is part of a cycle a different token set is created. As a consequence tokens belonging to such a set cannot be synchronized with any other token; however, to exit a cycle XOR-joins can be used.

\[
\text{OrSplitGateTransition}(\text{node}) = \text{WorkflowTransition}(\text{node})
\]

where

\[
\begin{align*}
\text{let}\{\text{in}\} &= \text{inArc}(\text{node}) \\
\text{let}\ O &= \text{select}_\text{produce}(\text{node}) \\
\text{let}\ t &= \text{firingToken}(\{\text{in}\}) \\
\text{CTLCOND}(\text{node}) &= \text{Enabled}(\text{in}, \text{tokenSet}(t)) \\
\text{CTLOP}(\text{node}, O) &= \text{CONSUME}(t, \text{in}) \\
\text{CONSUMESYNCALL}(\{(t, i) \mid i \in \text{AllJoinArc}(o)\ \text{for some}\ o \in \text{outArc}(\text{node})\}) \\
\text{PRODUCESYNCALL}(\{(\text{orSplitToken}(t, o), i) \mid i \in \text{AllJoinArc}(o), o \in O\}) \\
\text{forall}\ o \in O \\
&\quad \text{if}\ \text{cycle}(o) \\
&\quad \text{PRODUCE}(\text{orSplitToken}(t, o), o) \\
&\quad \text{tokenSet}(\text{orSplitToken}(t, o)) := \text{genTokenset}(t, o) \\
&\quad \text{else} \\
&\quad \text{PRODUCE}(\text{orSplitToken}(t, o), o) \\
&\quad \text{tokenSet}(\text{orSplitToken}(t, o)) := \text{tokenSet}(t)
\end{align*}
\]

We apply the same changes to the **AndSplitGateTransition** submachine. Since token sets can no longer pass splits if the outgoing token might return to the split, \(\text{AllJoinArc}\) need no longer refer to all reachable incoming edges of joins. Rather, we need it to refer to those incoming edges of joins that are reachable without creating a new token set. Because we use \(\text{AllJoinArc}\) to block joins with synchronization tokens, this modification translates the independence of cycles that we gained by creating new token sets at the cyclic edges of splits to the blocking discipline. The definition of \(\text{AllJoinArc}(o)\) is refined to refer to exactly those incoming edges of joins in the workflow that are reachable from \(\text{target}(o)\) via a path that contains no outgoing cyclic edge of a split.

The refined model is again a conservative extension of the previous one and thus “backwards-compatible” with the BPMN standard. In fact, in the acyclic case, the token set created by the start event will be the only one that is active in a process instance. Because all tokens belong to this token set, the refined ASM behaves just like the one in the last section. If there are cycles, the behaviour is ‘defined’ in [8] by some examples of cyclic workflows with a suggested mapping to BPEL.
The workflow depicted in Fig. 3 is the most complex cyclic example in the standard. The reader will identify the split nodes by their unique incoming arc and the join nodes by their unique outgoing arc. Note that all the splits are XOR-splits, so there is only one token in the cycle at any given time and the intuitive semantics of the workflow is quite obvious. In our model, tokens can enter the cycle because the join leading to the “Configure Product” task can only be reached from outside of the cycle (starting it), or via cyclic edges of splits (from inside the cycle). This means that the only incoming edge of the initial join that contains a synchronization token corresponding to a token that just triggered the “Assemble Components” is the one on which the very same token was placed after it triggered “Assemble Components”. In a similar manner, the first join is enabled when “Test Level 1” determines that “Test Level 2” need not be conducted and the control flow loops directly back to that join.

The BPMN standard allows what is named “Infinite Loop” [8, p.200], better called “Closed Loop”. A closed loop is a cycle without any split. Tokens that enter a closed loop are forever lost to the rest of the workflow. In our model, this leads to a deadlock, because each token entering the closed loop will have a synchronization copy of itself placed on the incoming edge of the initial join that loops back from the cycle. It is hard to imagine a sensible real-world example that contains a closed loop (the BPMN standard document admits this). Banning closed loops from workflows is thus not a serious restriction, especially since infinitely looping cycles are still possible as long as they are not closed.

This model for OR-joins is furthermore refined in [24], extending the refinement technique introduced here for synchronization tokens, to the more general case where multiple tokens can be present in a cycle with multiple entry and exits points. The following properties are proved:

- Acyclic workflow diagrams are deadlock free.
- Workflow diagrams with cycles, but without sync-splits or sync-joins and without closed loops, are deadlock free.
- A class of stratified workflows is defined which is proved to be free of deadlocks (if there are no closed loops).
- An algorithm is defined for arbitrary workflow diagrams such that if the algorithm yields output “deadlock free”, then the workflow has no deadlocks.
- Acyclic workflow diagrams terminate and each flow object fires at most once.
- Progress in deadlock free cyclic workflow diagrams.

A simulator has been derived from the model presented here, which makes the specification executable.
5 Concluding Remarks

Based upon the definitions provided in this paper for various OR-join semantics, one can apply any rigorous technique to the validation and verification of business process diagrams containing OR-joins. For example the simulator developed in [24] for the visualization of BPMN workflows has been used for the validation of the definitions in this paper; as a verification example one finds there also a proof that stratified workflows are deadlock free. There is no limitation to tool sets of specific modeling frameworks. One can use the definitions to design business process diagram schemes and their instantiations in parallel with proving properties of interest for them, using the feature-based approach illustrated in [2] and choosing appropriate tools to support theorem proving, model checking, static analysis etc.

References


Modeling Workflows, Interaction Patterns, Web Services and Business Processes: The ASM-Based Approach

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 survey the use of the Abstract State Machines (ASM) method for a rigorous foundation of modeling and validating web services, workflows, interaction patterns and business processes. We show in particular that one can tailor business process definitions in application-domain yet rigorous terms in such a way that the resulting ASM models can be used as basis for binding contracts between domain experts and IT technologists. The method combines the expressive power and accuracy of rule-based modeling with the intuition provided by visual graph-based descriptions. We illustrate this by an ASM-based semantical framework for the OMG standard for BPMN (Business Process Modeling Notation). The framework supports true concurrency, heterogeneous state and modularity (compositional design and verification techniques). As validation example we report some experiments, carried out with a special-purpose ASM simulator, to evaluate various definitions proposed in the literature for the critical OR-join construct of BPMN\(^3\).

1 Introduction

Over the last five years the Abstract State Machines (ASM) method has been used successfully in various projects concerning modeling techniques for web services, workflow patterns, interaction patterns and business processes.

An execution semantics for (an early version of) the Business Process Execution Language for Web Services (BPEL) has been provided in terms of ASMs in \([19,23]\) and has been reused in \([17,18]\). In \([4]\) one finds a survey of recent applications of the ASM method to design, analyze and validate execution models for service behavior mediation \([3]\), service discovery \([2,20]\) and service composition techniques \([22]\), three typical themes concerning Service Oriented Architectures (SOAs). In \([22]\) multi-party communication is viewed as an orchestration problem (“finding a mediator to steer the interactions”). A systematic analysis, in terms of ASMs, of complex communication structures built from basic service interaction patterns has been carried out in \([5]\). The workflow patterns collected in \([27,24]\), which are widely considered in the literature as paradigms for business process control structures, have been shown in \([10]\) to be instances of eight (four sequential and four parallel) basic ASM workflow schemes.

Recently, we have adopted the ASM method for a systematic study of business process modeling techniques \([14,13,12]\). As authoritative reference for basic concepts and definitions we have chosen the OMG standard for BPMN \([15]\), which has been defined to reduce the fragmentation of business process modeling notations and tools. In the following we describe the salient methodological features of this work and report our experience in applying the ASM framework \([11]\) to provide a transparent accurate high-level definition of the execution semantics of the current BPMN standard (version 1.0 of 2006).

The paper is organized as follows. In Sect.\(^2\) we explain how ASM models for business processes can serve as ground model and thus as basis for a precise software contract, allowing one to address the correctness question for a business process description with respect to the part of the real-world

\(^1\) The work of the first author is supported by a Research Award from the Alexander von Humboldt Foundation (\textit{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.
it is supposed to capture. In Sect. 3 we list the main methodological principles which guided us in defining a succinct modularized ASM that constitutes an abstract interpreter for the entire BPMN standard. In Sect. 4 we formulate the ASM rule pattern that underlies our feature-based description of specific workflow behaviors. In Sect. 5 we show how this scheme can be instantiated, choosing as example BPMN gateways. In Sect. 6 we illustrate by a discussion of the critical BPMN OR-join construct how one can put to use an appropriate combination of local and global state components in ASMs. We report here some results of an experimental validation, performed with a special-purpose ASM interpreter 25 that integrates with current graphical visualization tools, of different definitions proposed for the OR-Join in the literature. In Sect. 7 we point to some directly related work and research problems.

2 Building ASM Ground Models for Business Processes

To guarantee that software does what the customer expects it to do involves first of all to accurately describe those expectations and then to transform their description in a controlled way to machine code. This is a general problem for any kind of software, but it is particularly pressing in the case of software-driven business process management, given the large conceptual and methodological gap between the business domain, where the informal requirements originate, and the software domain, where code for execution by machines is produced. The two methodologically different tasks involved in solving the problem have been identified in 9 as construction of ground models, to fully capture the informal requirements in an experimentally validatable form, and their mathematically verifiable stepwise detailing (technically called refinement) to compilable code.

Ground models represent accurate “blueprints” of the piece of “real world” (here a business process) one wants to implement, a system reference documentation that binds all parties involved for the entire development and maintenance process. The need to check the accuracy of a ground model, which is about a not formalizable relation between a document and some part of the world, implies that the model is described in application domain terms one can reliably relate to the intuitive understanding by domain experts of the involved domain phenomena. Ground models are vital to reach a firm understanding that is shared by the parties involved, so that a ground model has to serve as a solid basis for the communication between the (in our case three) parties: 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). We refer for a detailed discussion of such issues to 9 and limit ourselves here to illustrate the idea by three examples of a direct (i.e. coding-free, abstract) mathematical representation of business process concepts in ASM ground models. The examples define some basic elements we adopted for the abstract BPMN interpreter in 13.

Example 1. Most business process model notations are based on flowcharting techniques, where business processes are represented by diagrams at whose nodes activities are executed and whose arcs are used to contain the information on the desired execution order (so-called control information). We therefore base our BPMN model on an underlying graph structure, at whose nodes ASM rules are executed which express the associated activity and the intended control flow.

Furthermore, usually the control flow is formulated using the so-called token concept, a program counter generalization known from the theory of Petri nets. The idea is that for an activity at a target node of incoming arcs to become executable, some (maybe all) arcs must be Enabled by a certain number of tokens being available at the arcs; when executing the activity, these tokens are Consumed and possibly new tokens are Produced on the outgoing arcs. This can be directly expressed using an abstract dynamic function token associating (multiple occurrences of) tokens—elements of an abstract set Token—to arcs

\[ \text{token} : \text{Arc} \to \text{Multiset}(\text{Token}) \]

2 In programming language terms one can understand \( f(a_1, \ldots, a_n) \) for a dynamic function \( f \) as array variable.
The use of an abstract predicate *Enabled* and abstract token handling machines *CONSUME* and *PRODUCE* allows us to adapt the token model to different instantiations by a concrete token model. For example, a frequent understanding of *Enabled* is that of an atomic quantity formula, stating that the number of tokens currently associated to an arc *incoming* into a given node is at least a quantity \( inQty(in) \) required at this arc.

\[
\text{Enabled}(in) = (\| \text{token}(in) \| \geq inQty(in))
\]

With such a definition one can also specify further the abstract control related operations, namely to *CONSUME* (\( inQty(in) \) many occurrences of) a token \( t \) on \( in \) and to *PRODUCE* (\( outQty(out) \) many occurrences of) \( t \) on an arc *outgoing* from the given node.

\[
\begin{align*}
\text{CONSUME}(t, in) &= \text{DELETE}(t, inQty(in), \text{token}(in)) \\
\text{PRODUCE}(t, out) &= \text{INSERT}(t, outQty(out), \text{token}(out))
\end{align*}
\]

We express the *data* and *events*, which are relevant for an execution of the activity associated to a node and belong to the underlying database engine respectively to the environment, by appropriate ASM locations: so-called controlled (read-and-write) locations for the data and monitored (only read) locations for the events. Any kind of whatever complex value an application needs for data or events is allowed to be stored in an ASM location, directly, corresponding to the given level of abstraction, avoiding any encoding a later refinement to implementable data structures may require.

This approach allows us to combine the visual appeal of graph-based notations with the expressive power and simplicity of abstract-state and rule-based modeling: we can paraphrase the informal explanations in the BPMN standard document of “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 by corresponding ASM rules, which address issues the graphical notation does not clarify. More generally speaking, (asynchronous) ASMs can be operationally understood as extension of (locally synchronous and globally asynchronous [21]) Finite State Machines to FSMs working over abstract data. Therefore a domain expert, when using graphical design tools for FSM-like notations, can reason about the graphical elements in terms of ASMs whenever there is some need for an exact reference model to discuss semantically relevant issues.

**Example 2.** In business process design it is usual to distinguish between a *business process* (the static diagram) and its *instances* (with specific token marking and underlying data values). This distinction is directly reflected in the ASM model in terms of instantiations of the underlying parameters, which appear abstractly but explicitly in the ASM model. For example a token is usually characterized by the process ID of the process instance \( pi \) to which it belongs (via its creation at the start of the process instance), which allows one to distinguish tokens belonging to different instances of one process \( p \). It suffices to write \( token_{pi} \) to represent the current token marking in the process diagram instance of the process instance \( pi \) a token belongs to. In this way \( token_{pi}(arc) \) denotes the *token* view of process instance \( pi \) at \( arc \), namely the multiset of tokens currently residing on \( arc \) and belonging to process instance \( pi \). BPEL uses this for a separation of each process instance by a separate XML document.

Correspondingly one has to further detail the above predicate *Enabled* by the stipulation that only tokens belonging to one same process instance have to be considered:

\[
\text{Enabled}(in) = (\| \text{token}_{pi}(in) \| \geq inQty(in) \text{ for some } pi)
\]

The reader will notice that the use of abstract *INSERT* and *DELETE* operations in defining the macros *PRODUCE* and *CONSUME* for tokens, instead of directly updating *token(a, t)*, comes handy: it makes the macros usable in a concurrent context, where multiple agents, belonging to multiple process instances, 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.

Thus the ASM model of the given business process represents the scheme of the process, statically defined by its rules; its instances are the scheme instantiations obtained by substituting the parameters by concrete values belonging to the given process instance \( pi \). In accordance with common practice, one can and usually does suppress notationally the process instance parameter \( pi \), as we did when explaining the function \( token \) above, as long as it is clear from the context or does not play a particular role.

**Example 3.** This example is about the need for global data or control structures in various business process constructs, e.g. synchronization elements owned by cooperating process agents or more generally speaking data shared by local processes. They can be directly expressed in terms of global locations of possibly asynchronous ASMs, thus avoiding detours one has to invent in frameworks where transitions can express only local effects. For an illustration we refer again to the definition of the BPMN standard in [15]. It uses a predominantly local view for task execution and step control, although some constructs such as splits and joins, multi-instance processes, gotos (called links), and sub-processes are bound by context integrity constraints.

For example splits can be intimately related by such integrity constraints to joins and thus their execution is not free of side (read: not local) effects. For an illustration see the discussion of the OR-join gateway construct of BPMN in Sect. 6.

Another example are data dependencies among different processes, whose description in [15] seems to be relegated to using associations, but really need global or shared locations to appropriately represent their role for the control flow.

3 Separation of Different Concerns

For design and analysis of business processes it turned out to be crucial that the ASM method supports to first explicitly separate and then smoothly combine the realization of different concerns, based upon appropriate abstractions supporting this form of modularization. We list some of the main separation principles, which we have used with advantage for the definition of the execution semantics for BPMN by ASMs.

**Separation Principle 1.** This principle is about the separation of behavior from scheduling. To cope with the distributed character of cooperating business processes, one needs descriptions that are compatible with various strategies to realize the described processes on different platforms for parallel and distributed computing. This requires the underlying model of computation to support most general scheduling schemes, including true concurrency.

In general, in a given state of execution of a business process, 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 omymous \( Enabled \)ness predicate for arcs) if at least one of its associated rules is Fireable at this node in this state.

We separate the description of workflow behavior from the description of the underlying scheduling strategy in the following way. We define specific business process transition rules, belonging to a set say \( WorkflowTransition \) of such rules, to describe the behavioral meaning of any workflow construct associated to a node. Separately, we define an abstract scheduling mechanism, to choose at each moment an enabled node and at the chosen node a fireable transition, by two not furthermore specified selection functions, say \( selectNode \) and \( selectWorkflowTransition \) defined over the sets \( Node \) of nodes respectively \( WorkflowTransition \). These functions determine how to choose an

---

3 We treat the fireability of a rule (by an agent) as an abstract concept, because its exact interpretation may vary in different applications. For business process diagrams it clearly depends on the \( Enabled \)ness of the incoming arcs related to a rule at the given node, but typically also on further to be specified aspects, like certain events to happen, on the (degree of) availability of needed resources, etc.
enabled node and a fireable workflow transition at such a node for its execution. We then can combine behavior and scheduling by a rule scheme WorkflowTransitionInterpreter, which expresses how 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.

\[
\text{WorkflowTransitionInterpreter} = \\
\text{let } node = \text{select}_{\text{Node}}(\{ n \mid n \in \text{Node} \text{ and } \text{Enabled}(n) \}) \\
\text{let } rule = \text{select}_{\text{WorkflowTransition}}(\{ r \mid r \in \text{WorkflowTransition} \text{ and } \text{Fireable}(r, node) \})
\]

Separation Principle 2. The second principle is about the separation of orthogonal constructs. To make ASM workflow interpreters easily extensible and to pave the way for modular and possibly changing workflow specifications, we adopted a feature-based approach, where the meaning of workflow concepts is defined elementwise, construct by construct. For each control flow construct associated to a node we provide a dedicated rule (or set of rules) WorkflowTransition(node), belonging to the set WorkflowTransition in the WorkflowTransitionInterpreter scheme of the previous example, which abstractly describe the operational interpretation of the construct. We illustrate this in Sect. 5 by the ASM rules defining the execution behavior of BPMN gateways, which can be separated from the behavioral description of BPMN event and activity nodes.

Another example taken from BPMN is the separation of atomic tasks from non-atomic subprocesses and from activities with an iterative structure. BPMN distinguishes seven kinds of tasks:

\[
\text{TaskType} = \{ \text{Service, User, Receive, Send, Script, Manual, Reference, None} \}
\]

These task types are based on whether a message has been sent or received or whether the task is executed or calls another process. The execution semantics for task nodes is given by one ASM rule (scheme) [13], which uses as interface abstract machines to Send or Receive messages and to Call or Execute processes.

A third example from the BPMN standard is the separation of cyclic from acyclic processes, which we use for the discussion of the OR-join gateway in Sect. 6.

Separation Principle 3. The third principle is the separation of different model dimensions like control, events, data and resources. Such a separation is typical for business process notations, but the focus of most of these notations on control (read: execution order, possibly influenced also by events) results often in leaving the underlying data or resource features either completely undefined or only partly and incompletely specified. 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. We illustrate this in Sect. 4 by the four components for data, control, events and resources in WorkflowTransition, which constitute four model dimensions that come together in the ASM scheme for workflow interpreter rules. These four components are extensively used in the BPMN standard, although the focus is on the control flow, which is represented by the control flow arcs, relegating interprocess communication (via message flow arcs between processes) and data conditions and operations to minor concerns. For an enhanced specification of interprocess communication see the orchestration of processes in [28].

Separation Principle 4. The fourth principle, whose adoption helps to reduce the description size of abstract models, is the separation of rule schemes and concrete rules, where the concrete rules may also be specialized rule schemes. It exploits the powerful abstraction mechanisms ASMs offer for both data and operations, whether static or dynamic. We illustrate this by the ComplexGateTransition in Sect. 5.1, a scheme from which one can easily define the behavior of the other BPMN gateways by instantiating some of the abstractions (see Sect. 5.2).
Separation Principle 5. The fifth example is about the separation of design, experimental validation and mathematical verification of models and their properties. In Sect. 6 we illustrate an application of this principle by an analysis of the OR-join gateway, where for a good understanding of the problem one better separates the definition of the construct from the computation or verification of its synchronization behavior. Once a ground model is defined, one can verify properties for diagrams, separating the cases with or without cycles and in the former case showing which cycles in a diagram are alive and which ones may result in a deadlock. In addition, the specialized ASM workflow simulator [25] allows one to trace and to experimentally validate the behaviour of cyclic diagrams.

The principle goes together with the separation of different levels of detail at which the verification of properties of interest can take place, ranging from proof sketches over traditional or formalized mathematical proofs to tool supported proof checking or interactive or automated theorem proving, all of which can and have been used for ASM models (see [11] Ch.8,9 for details).

Separation Principle 6. This principle is about the separation of responsibilities, rights and roles of users of BPMN diagrams. To represent different roles of users BPMN diagrams can be split into so-called pools, between which messages can be exchanged. Furthermore user actions can be separated by so-called swimlanes. Such a separation of user actions depending on the user’s role within a diagram is supported in a natural way by the ASM concept of rule executing agents: one can associate different and even independent agents to sets of user rules; moreover these agents could be supervised by a user superagent coming with his own supervising rules, which leads to more general interaction patterns than what is foreseen by the BPMN standard (see [5]).

In the next section we show how from a combination of the separation principles formulated above one can derive an orthogonal high-level interpretation of the basic concepts of BPMN.

4 The Scheme for Workflow Interpreter Rules

For every workflow or BPMN construct associated to a node, its behavioral meaning can be expressed by a guarded transition rule $\text{WorkflowTransition}(\text{node}) \in \text{WorkflowTransition}$ 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. This allows one to instantiate them by further detailing the guards (expressions) respectively the submachines for the description of concrete workflow transitions.\(^4\)

\[\text{WorkflowTransition}(\text{node}) = \]
\[\text{if EventCond}(\text{node}) \text{ and CtrlCond}(\text{node}) \]
\[\text{and DataCond}(\text{node}) \text{ and ResourceCond}(\text{node}) \text{ then}
\]
\[\text{DataOp}(\text{node}) \text{ and CtrlOp}(\text{node}) \]
\[\text{EventOp}(\text{node}) \text{ and ResourceOp}(\text{node}) \]

\(^4\) 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.
5 Instantiating WorkflowTransition for BPMN Gateways

In this section we instantiate WorkflowTransition for BPMN gateways, nodes standing for one of the three types of BPMN flow objects. The other two types are event and activity nodes, whose behavior can be described by similar instantiations, see \[13\] for the details. We start with the rule for so-called complex gateway nodes, from which the behavior of the other BPMN gateway constructs can be defined as special cases.

5.1 ComplexGateTransition

Gateways are used to describe the convergence (also called merging) and/or divergence (also called splitting) of control flow, in the sense that tokens can ‘be merged together on input and/or split apart on output’ \[15\], p.68]. For both control flow operations one has to determine the set of incoming respectively outgoing arcs they are applied to at the given node. The particular choices depend on the node, so that we represent them by two abstract selection functions, namely to

- selectConsume the incoming arcs where tokens are consumed,
- selectProduce the outgoing arcs where tokens are produced.

Both selection functions come with constraints: selectConsume is required to select upon each invocation a non-empty set of enabled incoming arcs, whose firingTokens are to be consumed in one transition, selectProduce is constrained to select upon each invocation a non-empty subset of outgoing arcs satisfying an associated OutCond(o). On these arcs complexGateTokens are produced, whose particular form may depend on the firingTokens. We skip that in addition, as (part of) DataOp(node), multiple assignments may be ‘performed when the Gate is selected’ \[15\], Table 9.30 p.86] (read: when the associated rule is fired).

ComplexGateTransition(node) =
let
 I = selectConsume(node)
 O = selectProduce(node)
 in WorkflowTransition(node, I, O)
where
CtlCond(node, I) = (I ≠ ∅ and forall in ∈ I Enabled(in))
CtlOp(node, I, O) =
  if O ≠ ∅ and O ⊆ \{ o ∈ outArc(node) | OutCond(o) \}
  then
    ProduceAll(\{ (complexGateToken(firingToken(I), o), o) | o ∈ O \})
    ConsumeAll(\{ (in_i, o_i) | 1 ≤ i ≤ n \})
   where
    [in_1, ..., in_n] = firingToken(I), [o_1, ..., o_n] = O
  else
    CtlOp(node, I, O)

A function firingToken(A) is used to express a structural relation between the consumed incoming and the produced outgoing tokens, as described in \[15\], p.35. It is assumed to select for each element a of an ordered set A of incoming arcs some of its token(a) 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, ..., a_n]) = [t_1, ..., t_n] denotes that t_i is the token selected to be fired on arc a_i.
5.2 Instantiating ComplexGateTransition

The BPMN standard defines and names also special gateways, which can all be obtained by specializing the selection functions in ComplexGateTransition. To describe these instantiations more clearly, we assume without loss of generality that these special gateways never have both multiple incoming and multiple outgoing arcs. Thus the so-called split gateways have one incoming and multiple outgoing arcs, whereas the so-called join gateways have multiple incoming and one outgoing arc.

For AND-split and AND-join gateway nodes, \( \text{select}_{\text{Produce}} \) and \( \text{select}_{\text{Consume}} \) are required to yield all outgoing resp. all incoming arcs.

For OR-split nodes two cases are distinguished: \( \text{select}_{\text{Produce}} \) chooses exactly one (exclusive case, called XOR-split) or at least one outgoing arc (called inclusive OR or simply OR-split). For the exclusive case a further distinction is made depending on whether the decision is ‘data-based’ or ‘event-based’, meaning that \( \text{OutCond}(o) \) is a \( \text{DataCond}(o) \) or an \( \text{EventCond}(o) \). For both cases it is required to select the first \( \text{out} \in \text{outArc}(\text{node}) \), in the given order of gates, satisfying \( \text{GateCond}(\text{out}) \).

Similarly also for OR-join nodes two versions are distinguished, an exclusive and data-based one—the event-based XOR is forbidden by the standard to act only as a Merge—and an event-based inclusive one. In the latter case \( \text{select}_{\text{Consume}} \) is required to yield a subset of the incoming arcs with associated tokens ‘that have been produced upstream’ [15, p.80], but no indication is given how to determine this subset, which is a synchronization problem. We discuss this very much disputed issue further in the next section.

6 OR-Join Gateway: Global versus Local Description Elements

The OR-join concept is present in many workflow and business process modeling languages and is used with different understandings advocated in the literature, in different commercial workflow systems and by different users. Part of this situation stems from the fact that in dealing with the OR-join concept, often two things are mixed up that should be kept separate, namely a) how the intended meaning of the concept is defined (question of semantics) and b) how properties of interest for the construct (most importantly its fireability in a given state) can be computed, validated (at run time) or verified (at design time) (question of computation, validation and verification methods).

It could be objected that an algorithm to compute the fireability of the OR-join rules defines the crucial synchronization property and thus the semantics of the OR-join. Speaking in general terms this is true, but then the question is whether there is agreement on which algorithm to use and whether the algorithm is understandable enough to serve as a behavioral specification the business process expert can work with. However, looking at the literature there seems to be no agreement on which algorithm should be used and the complexity of the proposed ones makes them unfit to serve as satisfactory semantical specification for the workflow practitioner.

The semantical issue disputed in the literature is the specification of the \( \text{select}_{\text{Consume}} \) functions, which incorporate the critical synchronization conditions. \( \text{select}_{\text{Consume}}(\text{node}) \) 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 \( \text{node} \). Unfortunately, most proposals for an OR-join semantics in one way or the other depend on the framework used for the definition. This is particularly evident in the case of Petri-net-based definitions, where, to circumvent the restrictions imposed by the local nature of what a Petri net transition can express, either the diagrams are restricted (to the possible dislike of a business process practitioner) or ad hoc extensions of Petri nets are introduced that are hard to motivate in application domain terms (see for example [33, 31, 32]). A side problem is that the BPMN standard document seems to foresee that the function is dynamic (run-time determined), since 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]

We refer to [12] for a detailed discussion of OR-join variations and ways to define and compute the underlying synchronization functions \textit{select} and \textit{consume}. We restrict our attention here to report some experiments Ove Soerensen has made with various alternatives we considered to come up with a practically acceptable definition that could serve for the standard, in particular in connection with diagrams that may contain cycles. For this purpose Soerensen has built a specialized ASM workflow simulator [25] that is interfaced with standard graph representation tools, so that the token flow and the unfolding of a diagram cycle triggered by applying ASM OR-join rules can be visualized.

One alternative we considered is to a) pass at runtime every potential synchronization request from where it is originated (a split gateway node) to each downstream arc that enters a join gateway node and to b) delete this request each time the synchronization possibility disappears due to branching. Assume for the moment that the given diagram contains no cycles and assume without loss of generality that there is a unique start node. Then it suffices to operate the following refinement on our BPMN model.

- **Split gate transition refinement** When due to an incoming token \( t \) at a split node a new token \( t.o \) is produced on an arc \( o \) outgoing node, a computation path starts at \( o \) that may need to be synchronized with other computation paths started simultaneously at this node, so that also a synchronization copy is produced and placed on each downstream arc that enters a join node, i.e. an arc entering a join node to which a path leads from \( o \). We denote the set of these join arcs by \textit{AllJoinArc}(o). Simultaneously the synchronization copy of \( t \) is deleted from all such arcs that are reachable from node.

- **Join gate transition refinement** We consume the synchronization tokens that, once the to-be-synchronized tokens have been fired, have served their purpose, and produce new synchronization tokens for the tokens the join produces. To \textit{CtlCond}(node, I) we add the synchronization condition that \( I \) is a synchronization family at node, which means a set of incoming arcs with non-empty syncToken sets such that all other incoming arcs (i.e. those not in \( I \)) have empty syncToken set (read: are arcs where no token is still announced for synchronization so that no token will arrive any more (from upstream) to enable such an arc).

It is not difficult to formulate this idea as a precise refinement (in the sense of [8]) of our ASMs for BPMN split and join rules (see [12]). To extend this approach to the case of diagrams with cycles (more generally subprocesses), one can refine the \textit{AllJoinArc} function to yield only arcs of join nodes up to and including the next subprocess entry node; inside a subprocess \textit{AllJoinArc} is further restricted to only yield join nodes that are inside the subprocess. The production of synchronization tokens by the transition rule for join gate nodes that enter a subprocess is postponed to the exit node rule(s) of the subprocess.

There are obviously various other possibilities, with all of which one can experiment using the work that will be reported in [25].

7 Related and Future Work

There are two specific papers we know on the definition of a formal semantics of a subset of BPMN. In [16] a Petri net model is developed for a core subset of BPMN which however; it is stated there that due to the well-known lack of high-level concepts in Petri nets, this Petri net model “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 [30] 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

\footnote{In Soerensen’s tool this is realized by spanning a new diagram copy of the subprocess.}
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. The execution semantics for BPMN defined in \[13\] covers every standard construct and is defined in the form of \texttt{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 assign a precise mathematical 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, but allowing one to use, appropriate mechanical (theorem proving or model checking) techniques.

In \[29\] an inclusion of process interaction and resource usage concerns is advocated for the forthcoming extension BPMN 2.0 of BPMN. It could be worth to investigate how the ASM models defined in \[5\] for the interaction patterns in \[1\] can be included into the current ASM model for BPMN, extending the current communication means in BPMN—event handling, message exchange between pools and data exchange between processes—to richer forms of interaction between multiple processes. Also a rigorous analysis of scheduling and concurrency mechanisms would be interesting, in particular in connection with concerns about resources and workload balancing that play a crucial role for efficient implementations.

The feature-based definition of workflow concepts in this paper is an adaptation of the method used in a similar fashion in \[26\] 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 \[7\], see \[6\] for the relation between the two methods.

References


ASM Foundations of Database Management

Bernhard Thalheim\textsuperscript{1} and Klaus-Dieter Schewe\textsuperscript{2}

\textsuperscript{1} Christian Albrechts University Kiel, Department of Computer Science
Olshausenstr. 40, D-24098 Kiel, Germany
\textsuperscript{2} Massey University, Information Science Research Centre
Private Bag 11 222, Palmerston North, New Zealand
thalheim@is.informatik.uni-kiel.de k.d.schewe@massey.ac.nz

Abstract. Database structuring is well understood since decades. The operating of databases has been based in the past on temporal logics and did not yet get an easy to understand formal underpinning. Therefore, conceptions like transaction and recovery are mainly discussed at the logical or operational level. This paper shows that database structuring and functionality can be defined within a uniform language. We base database semantical on the operational semantics of abstract state machines (ASM). This uniform mechanism allows to define the structuring, the functionality, the distribution and the interactivity of a database system in a way that supports abstract consideration at various layers of abstraction, that supports refinement of specifications to more detailed ones and that support proof of properties.

1 Adequacy and Deficiencies of Database Technology

1.1 Strength and Weaknesses of Database Technology

Database systems are currently broadly used for support of data-intensive services. These broad usage is based on advantages such as the following:

Consistent storage of data: Data are uniquely stored in the most actual version.
Each user gets the same data. Inconsistency can be avoided. Furthermore, redundancy can be reduced and standards can be enforced.

Multi-user support: Data can be consistently shared among different users. Also, conflicting requirements can be balanced. Security is enforced by restricting and managing access to data. Data can be consistently distributed within a network.

Integration into component-ware: Currently, database systems are turned into middle-ware components in information-intensive applications. Database operating is based on the transaction paradigm. A transaction is a logical unit of work. Database systems are designed to support transactions.

Nevertheless, database engines do not completely support complex applications such as internet services, real-time applications, stream information systems and web information systems.
Content and information instead of data: With the development of more complex application sophisticated support for various abstraction levels on data is required. Instead of raw or micro-data users want to retrieve condensed data or macro-data and to work on them.

Transaction semantics: The transaction approach is based on atomicity, consistency, isolation and durability requirements on transaction application. User action are sometimes not atomic and require sophisticated support for intermediate actions. The isolation level may vary over time. The are different degrees for durability.

View support: Each object in the database needs to be identifiable. Query languages allow to construct macro-data from the micro-data of the objects contained in the database. These macro-data may be composed into complex derived objects. Due to the query language some of them may be not identifiable. Further, their connection to the micro-data may be not injective. Therefore, data manipulation on macro-data cannot translated to data manipulation on micro-data.

Missing operational semantics: Semantical treatment of structuring is based on the predicate logic since the advent of the relational database model. Entity-relationship modelling can be based on a generalized predicate logic. Object-oriented models are often defined in a fuzzy manner without existing models. Moreover, database operation is still not well based.

In this section we show now that the weaknesses of database models and technology for internet application can be reduced by the ASM approach.

1.2 Well-Founded Structuring and Operating Transparency

Database systems are typically specified through a specification of database structuring and the assumption of canonical database operating. Structuring of databases is given by a signature of the database and a set of static integrity constraints. They form the schema of the database. The signature is used as an alphabet of a canonical first-order (or first-order hierarchical [13]) predicate logic. Static integrity constraints may be expressed as formulas of this logic. The functionality of a database system is defined on the basis of an algebra that is generically defined over the signature. This algebra is extended to a query algebra and to modification operations for the database system. Database management systems provide services that are assumed to be given whenever we are talking on database operating. Their behaviour and their operating is assumed to be canonically given.

This assumption might be appropriate for the predesign or business user layer [5,13] of specification. The implementation independence of the business user layer and the operating transparency supports concentration application specific aspects of a database system. It is not appropriate for the conceptual layer since we need to consider database behaviour as well. This inappropriateness causes many problems for database programming and operating. This paper shows how the ASM approach can be used to overcome this gap.
Information systems extend database systems by a support for users, by interface systems and by content management. Web information systems [9, 12] augment classical information systems by modern web technologies. They aim in supporting a wide variety of users with a large diversity of utilisation stories, within different environments and with desires for personal web-enhanced work spaces. The development of WIS is therefore adding the user, story and interaction dimension to information systems development. So far information systems development could concentrate on the development of sophisticated structuring and functionality. Distributed work [10] has already partially been supported.

1.3 Necessities for Specification of Interactive Information Systems

Despite the presence of an active research community that studies conceptual models for information systems, interaction still lacks precise formal underpinnings. There are several approaches which can be generalized for development of a formal basis of interactive information services:

**Workflow and pattern:** Workflow approaches allow to combine dataflow and computation. Workflows can be constructed on the basis of basic processes by application of basic control constructors such as sequence, parallel split, exclusive choice, synchronization, and simple merge, by application of advanced branching and synchronization control constructors such as multiple choice, multiple merge, discriminator, n-out-of-m join, and synchronizing join and by application of structural control and state-based control constructors.

**Wegner’s interaction machines.** The approach is based on four assumptions: User rather observe machine behavior instead of having a complete understanding of machines I/O behavior. Modelling of systems cannot be entirely inductive. Instead of that co-induction needs to be used, at least for the earlier design phases. Users compete for resources. Thus, modelling includes an explicit specification of cooperation and competition. Interactive behavior cannot be entirely modelled on the basis of I/O behavior but rather in term of interaction streams.

We use the interaction machine approach [6] in order to reason formally on information services. An interaction machine [15] can be understood as Turing machine with multi-user dynamic oracles (MIM) or with single-user dynamic oracles (SIM). An interaction machine can be specified as follows [3]:

\[
\text{if condition}
\]

\[
\text{then state = Nextstate(state, database, input)}
\]

\[
\text{database = Modification(state, database, input)}
\]

\[
\text{output = output } \circ \text{ Out(state, database, input)}
\]

**Statechart diagrams:** The statechart approach has been proved to be useful for state-oriented modelling of database applications and more generally for use-case diagrams. Despite its usage in UML it has a precise semantics which can be easily integrated into ER modelling approaches [13].

**User interaction modelling:** Interaction involves several partners (grouped according to characteristics; group representatives are called ‘actors’), manifests itself in diverse activities and creates an interplay between these activities.
In order to develop usable websites the story of the application has to be neatly supported [9, 12, 10, 14]. Interaction modelling include modelling of environments, tasks and actors beside modelling of interaction flow, interaction content and interaction form [7].

1.4 Achievements of the ASM Approach

The abstract state machine (ASM) method nicely supports high-level design, analysis, validation and verification of computing systems:

- ASM-based specification improves industrial practice by proper orchestration of all phases of software development, by supporting a high-level modelling at any level of abstraction, and by providing a scientific and formal foundation for systems engineering. All other specification frameworks known so far only provide a loose coupling of notions, techniques, and notations used at various levels of abstraction.

By using the ASM method, a system engineer can derive a general application-oriented understanding, can base the specification on a uniform algorithmic view, and can refine the model until the implementation level is achieved. The three ingredients to achieve this generality are the notion of the ASM itself, the ground model techniques, and the proper treatment of refinement.

- Abstract state machines entirely capture the four principles [16] of computer science: structuring, evolution, collaboration, and abstraction.

This coverage of all principles has not been achieved in any other approach of any other discipline of computer science. Due to this coverage, the ASM method underpins computer science as a whole. We observe the following by comparing current techniques and technologies and ASM methods: ASM are running in parallel. Collaboration is currently mainly discussed at the logical or physical level. Evolution of systems is currently considered to be a hot but difficult topic. Architecture of systems has not yet been systematically developed.

- The ASM method is clearly based on a number of postulates restricting evolution of systems. For instance, sequential computation is based on the postulate of sequential time, the postulate of abstract state, and the postulate of bounded exploration of the state space. These postulates may be
extended to postulates for parallel and concurrent computation, e.g., by extending the last postulate to the postulate of finite exploration.

2 ASM Specification of Database Systems

Database systems are designed to operate in parallel. Operating of databases systems can be understood in a state-based approach.

2.1 ASM Specification of Databases and Database Structuring

The abstract state machine approach allows simple and refinable specification of parallel processes based on states and transitions.

The ASM signature $S$ is a finite collection of function names.

- Each function name $f$ has an arity, a non-negative integer.
- Nullary function names are called constants.
- Function names can be static or dynamic.
- Every ASM signature contains the static constants $undef$, $true$, $false$.

A database schema may be based on a collection of (predicative) functions representing the structures of the database. Typically such functions are dynamic. Static functions are those functions that do not change over time. Most generic database computation functions such as the aggregation functions can considered to be static. They are defined as static higher-order functions. Their concretisation to database functions may lead to a dynamic functions or may still be static. Database state functions are however dynamic.

The signature $S$ is the main component of alphabet for the logical language $\mathcal{L}_S$. We assume that this language is specified in the canonical way used in predicate logics. An ASM database schema $S$ is given by a signature $S$ and by a finite set $\Sigma$ of formulas from $\mathcal{L}_S$.

We restrict the consideration in this paper to the tuple (or product) constructor. The set, list, multiset, disjoint union, labelling and naming constructors can be treated in a similar way.

A database state $DB$ (or database instance) for the signature $S$ is a non-empty set $Val$ called the superuniverse of $DB$, together with an interpretation $f^{DB}$ of each function name $f$ in $S$.

- If $f$ is an n-ary function name of $S$, then $f^{DB} : Val^n \rightarrow Val$.
- If $c$ is a constant of $S$, then $c^{DB} \in Val$.
- The superuniverse $Val$ of the state $DB$ is denoted by $Val(DB)$.

Relations are functions that have the value $true$, $false$, $undef$

$(\bar{a} \in R \iff R(\bar{a}) = true)$.

\[ \text{dom}(f^{DB}) = \{(a_1, \ldots, a_n) \in Val(DB)^n \mid f^{DB}(a_1, \ldots, a_n) \neq undef\} \]

\[ \text{rel}(f^{DB}) = \{(a_1, \ldots, a_n) \in Val(DB)^n \mid f^{DB}(a_1, \ldots, a_n) = true\} \]

The superuniverse can be divided into subuniverses represented by unary relations. These unary relations form the basic data types under consideration. A
data type is typically given by a set of values and a set of (static) functions defined over these values.

A database state $\mathcal{DB}$ is a model of $\Sigma$ ($\mathcal{DB} \models \Sigma$) (or is called consistent database state) if $\mathcal{E}_{\mathcal{DB}}(\zeta) = \mathcal{E} = \text{true}$ for all variable assignments $\zeta$ for $\Sigma$. We typically consider only models. Since transactions may have intermediate database states that are not models we distinguish the two notions. The distinction between $\text{undef}$ and $\text{false}$ allows to separate for relational functions the case that an object does not belong to the database from the case that an object is known to be false.

2.2 Specification of Database Operating

Database state modifications can be described as changes to the dynamic functions. Changes we need to consider are either assigning a value $\text{undef}$, $\text{true}$, $\text{false}$ to a relational function or changes of the functions themselves. We thus may consider that a change can be given through a set of changes to the values for the functions of the domain of a function. This detailed view on the content of the database allows the introduction of database dynamics. Roughly speaking, we introduce memory cell abstractions (called locations) and consider the database to consist of those objects which locations evaluate to $\text{true}$. The functions of the database system can also be treated on the basis of locations.

A database location of $\mathcal{DB}$ is a pair

$$l = (f, (a_1, ..., a_n))$$

((relational) function, object). The value $\mathcal{DB}(l) = f_D(a_1, ..., a_n)$ is the content of the location $l$ in $\mathcal{DB}$.

The tuple $(a_1, ..., a_n)$ represents a potential object. The current value is given by a location $\mathcal{DB}(l)$ of the database system. The database consists of those objects $(a_1, ..., a_n)$ for which a relational function $f(a_1, ..., a_n)$ evaluates to $\text{true}$.

An modification $(l, u)$ to $\mathcal{DB}$ is the assignment of a value $u$ to a location $l$. For instance, an insert changes the value $\mathcal{DB}(l)$ at a location $l$ to $\text{true}$. The delete operation changes the value at a location $l$ to $\text{undef}$. A basic database update assigns $v$ to another location and changes the value of original location to $\text{undef}$.

A modification is called trivial if $v = \mathcal{DB}(l)$. A modification set consists of a set of modifications.

A modification set $U$ is consistent, if it has no clashing modifications, i.e., if for any location $l$ and all elements $v, w$ if $(l, v), (l, w) \in U$ then $v = w$.

The result of firing a consistent modification set $U$ is a new database state $\mathcal{DB} + U$

$$(\mathcal{DB} + U)(l) = \begin{cases} v & \text{if } (l, v) \in U \\ \mathcal{DB}(l) & \text{if there is no } v \text{ with } (l, v) \in U \end{cases}$$

for all $l$ of $\mathcal{DB}$.

For value-based types we define generic functions beyond insert, delete, update and evaluation functions such as average, min, max, sum. Additional functions $(F_f)_{f \in \mathcal{F}}$ are defined in the same manner on the type system. These functions can be used to construct a $\mathcal{S}$-algebra (e.g., operations such as projection,
join, selection, union, difference, intersection). We combine these functions into an S-algebra.

Databases change over time. We may represent the change history of a database by a sequence of database states. The changes are defined by an application of database systems operations to the current database state. We may require that any change applied to a model shall either transform the model to a new model or shall not be considered otherwise. This operating requirement leads to transaction semantics of database operating. We can use the logic $L_S$ for the definition of transition constraints.

A transition constraint consists of a pair of formulas $(\psi_{\text{pre}}, \psi_{\text{post}})$ from $L_S$. The transition constraint is valid for a database modification from $DB$ to $DB'$ if $DB \models \psi_{\text{pre}}$ and $DB' \models \psi_{\text{post}}$. Static integrity constraints from a finite set $\Sigma$ can be mapped to transition constraints $\bigwedge \alpha \in \Sigma \alpha$, $\bigwedge \alpha \in \Sigma \alpha$. Let $\Sigma_{\text{dynamic}}$ be the set of transition constraints.

A sequence of database states $DB_0, DB_1, \ldots, DB_{i+1} = \tau_A(DB_i), \ldots$ satisfying $\Sigma_{\text{dynamic}}$ is called a run of the database system. The run can be defined through $S$-algorithms $A$ or transformation rules that impose a one-step transformation $\tau_{DB}$ of a database state to its successor in the run.

A database program is given by a rule name $r$ of arity $n$ is an expression $r(x_1, \ldots, x_n) = P$ where $P$ is a database transition operation and the free variables of $P$ are contained in the list $x_1, \ldots, x_n$.

Database transition operations are either basic operations from the S-algebra or are constructed by inductively applying the following construction rules:

- Skip rule: $\text{skip}$
- Update rule: $f(s_1, \ldots, s_n) := t$
- Parallel execution rule: $P \text{ par } Q$
- Conditional rule: $\text{if } \phi \text{ then } P \text{ else } Q$
- Let rule: $\text{let } x = t \text{ in } P$
- For all rule: $\text{forall } x \text{ with } \phi \text{ do } P$
- Choose rule: $\text{choose } x \text{ with } \phi \text{ do } P$
- Sequence rule: $P \text{ seq } Q$
- Call rule: $r(t_1, \ldots, t_n)$

A database system consists of a database management system and of a number of databases. Let us consider only one database.

An abstract database system $M$ consists of a

- a signature $S_M$ of the database system that embodies the signature $S$ of the database,
- a set of initial states for $S_M$,
- a set of database programs,
- a distinguished program of arity zero called main program of the system.

We denote the current state of the database system by $DBS$ and by $DB$ the current state of the database. The transition operation $P$ yields the modification set $U$ in a state $DBS$ under the variable assignment $\zeta$: $\text{yields}^+(P, DBS, \zeta, U)$.

Semantics of transition operations defined in a calculus by rules:

<table>
<thead>
<tr>
<th>Premise1, ..., Premisen</th>
<th>Conclusion</th>
<th>Condition</th>
</tr>
</thead>
</table>
A query is an (open) $S$-formula. Moreover, a view is a query adorned by names (labels) for free variables. Due to our definitions, views are hierarchical too.

### 2.3 ASM Specification of Database System Behaviour

The database system state $D_{BS}$ is structured into four state spaces:

- input states $I$,
- output states $O$,
- DBMS states $E$,
- database states $D_{B}$.

The input states $I$ accommodate the database input to the database systems, i.e. queries and data. The output space $O$ allows to model the output data and error messages. The internal state space $E$ of the DBMS represents the DBMS states. The database content of the database system is represented in the database states $D_{B}$. The four state spaces are typically structured. This structuring is reflected in all four state spaces. For instance, if the database states are structured by a database schema then the input states can be structured accordingly.

The main database functions are modelled by modification, retrieval or internal control programs that are run in parallel:

- **Modification programs** allow to modify the database state if is enabled:
  
  if $I(req) \neq \lambda$ and $E(modify) = enabled$ and $I(req) \in Update$
  then $I(req) := \lambda$, $O(errMsg) := \ldots$, $D := \ldots$, $E := \ldots$

- **Retrieval rules** allow to retrieve the content of the database:
  
  if $I(req) \neq \lambda$ and $E(retrieve) = enabled$ and $I(req) \in Update$
  then $I(req) := \lambda$, $O(errMsg) := \ldots$, $O(answer) := \ldots$

- **DBMS control rules** allow to change the database state and the internal state of the database:
  
  if $E(DBMSstateChange)$ and $E(modify) = disabled$
  then $D := \ldots$, $E := \ldots$

The description can be extended in a similar fashion to sets of inputs instead of a singleton input.

The ASM programs are based on ASM rules

- `ModifyInput(request,DBMS_state, DB_state),`
- `ModifyOutput(request,DBMS_state, DB_state),`
- `ModifyDB(request,DBMS_state, DB_state),`
- `ModifyControl(request,DBMS_state, DB_state),`
- `RetrieveOutput(request,DBMS_state, DB_state),`
- `RetrieveDB(request,DBMS_state, DB_state),`
- `RetrieveControl(request,DBMS_state, DB_state),`
- `ControllerDBMS(DBMS_state, DB_state),` and
- `ControllerDB(DBMS_state, DB_state).`

These rules run in parallel. They express general DBMS functions for state modification:

- **modify** : $(req,s,d) \mapsto (_{errMsg,s',d'})$
- **retrieve** : $(req,s,d) \mapsto (_{answ,errMsg,s,d})$
- **controller** : $(s,d) \mapsto (_{s',d'})$

Modifications may cause deadlock. In order to overcome them the controller enables scheduling, recovery, and optimization. An modification can be imposed completely to the database for support of transaction semantics.
Summarizing we observe that ASM’s can be used for definition of operational semantics of database systems.

2.4 Co-Design of Structuring, Interaction and Behavior

Information systems design starts with database structuring and bases functionality on structuring. Typically, it uses various abstraction layers [13]: application domain layer for a rough specification of the application domain and its influence on the information system, requirements acquisition layer for the description of requirements such as main business data and business processes, business user layer or the predesign layer [5, 13] for the specification of business user objects and business workflows, conceptual layer for the conceptual description of structuring and functionality and implementation layer describing the implementation, i.e. code, SQL structures, interfaces and user views. Nowadays application tend to be distributed and components collaborate with each other [10]. Information systems provide services to users depending on their application tasks, portfolio, user stories and context [11, 12]. Therefore, we represent the four dimensions of information systems specification and their layers in Figure 2.4.

![Diagram of Abstraction Layers](image)

**Fig. 2.** The abstraction layer model for information systems specification

The structuring and functionality can be described as an abstract state machine. The stories of users, their profile, context and portfolio can be combined into an ASM-based description of the story space [2]. Interaction description can be based on notions developed in linguistics and in movie business [9].

According to the co-design approach we can distinguish two levels of consideration:

**Database system development layers:** The specification of database systems can be of different granularity and levels of detail. Typical development layers...
are motivation and requirements layers. A development layer (which is at the same time a database system run layer) is the conceptual layer. These layers display a database specification on different levels of detail and should be refinements of each other.

**Database systems operating layers:** Database operating is observed, maintained and administrated at various levels of detail: Business users have their own view onto the database application. These views may be different from each other. They are considered to be external views on the conceptual layer. The conceptual layer integrates all different aspects of database systems specification. The implementation layer is based on the logical and physical database schemata. Additionally, database systems functionality is added to the application on the basis of programs etc.

The ASM specification should match all different layers:

- **Level 1 (Point of view of business users):** Database system defined by three state space: input state, database state, output state. The *input state* is based on algebraic structure with ground terms defined on the values and the names. The *database state* is based on the (object-)relational structure with well-defined composition operators. The *output state* is a general database defined on the values and names.

- **Level 2 (Conceptual point of view):** Database systems are defined as an extension of level 1 by transactions, constraints, views and integrity maintenance.

- **Level 3-1 (Logical point of view):** The logical database system defined as an extension of level 2 by states of the database management system by the transaction and recovery engine, by the synchronization engine, the logging engine and the query translating engine.

- **Level 3-2 (Physical point of view):** The physical database system is defined as an extension of level 3-1 by specific functions of the DBMS.

- **Level 4 (DBMS point of view):** On level 4, the storage engine is modelled in detail with the buffers and the access engine.

This separation allows to model database systems at different levels of detail. Each higher level should be a refinement of the lower levels.

### 3 Faithful Refinement of Database Systems Specification

The co-design approach can be based on the abstraction layer model. We need a correctness criterion for the faithfulness of specifications among these different layers. We may distinguish between specifications at the requirements acquisition layer, at the business user layer and at the conceptual layer. (Static) integrity constraints typically limit the state space for the instances of database schema. Functionality can be provided within a framework for specifying the functions. The database system development process aims in stepwise refinement of structuring and functionality of a database system. This refinement process needs a formal underpinning. We can develop a theory of specification refinement for database applications based on ASM.
3.1 Refinements of Database Systems

Given two abstract database systems $\mathcal{M}$ and $\mathcal{M}^\ast$. The refinement of $\mathcal{M}_{DB}$ to $\mathcal{M}^\ast$ is based on

- a refinement of signatures $\mathbb{S}_M$ to $\mathbb{S}_{M^\ast}$ that associates functions and values of $\mathbb{S}_M$ with those on $\mathbb{S}_{M^\ast}$,
- a states of interest correspondence between the states of interest $DBS$ and $DBS^\ast$ defined over $\mathbb{S}_M$ and $\mathbb{S}_{M^\ast}$ correspondingly,
- abstract computation segments $DBS_1, \ldots, DBS_m$ on $\mathcal{M}$ and $DBS^\ast_1, \ldots, DBS^\ast_n$ on $\mathcal{M}$ and $\mathcal{M}^\ast$,
- locations of interest $(DB, DB^\ast)$ defined on $\mathbb{S} \times \mathbb{S}^\ast$ and an equivalence relation $\equiv$ on locations of interest.

$\mathcal{M}^\ast$ is a correct refinement of $\mathcal{M}$ if there for each $\mathcal{M}$-run $DBS^\ast_0, \ldots, DBS^\ast_k, \ldots$ there is an $\mathcal{M}$-run and sequences $i_0 < i_1 < \ldots$ and $j_0 < j_1 < \ldots$ such that $i_0 = j_0 = 0 \ DB_{i_k} \equiv DB^\ast_{j_k}$ for each $k$ and either

- both runs terminate and their final states are the last pair of equivalent states, or
- both runs and both sequences are infinite.

A refinement is called complete refinement if $\mathcal{M}_{DB}$ is a correct refinement of $\mathcal{M}^\ast_{DB}$ and $\mathcal{M}^\ast_{DB}$ is a correct refinement of $\mathcal{M}_{DB}$.

We distinguish between

- structure refinement that is based on a notion of schema equivalence and state equivalence,
- functionality refinement that is based on a notion of schema equivalence and a notion of coherence, and
- state refinement that uses the notion of equivalence and coherence.

3.2 Deriving Plans and Primitives for Refinement

The perspectives and styles of modelling rule the kind of refinement styles. As an example we consider structure-oriented strategies of development:

**Inside-out refinement:** Inside-out refinement uses the current ASM machine for extending it by additional part. These parts are hooked onto the current specification without changing it.

**Top-down refinement:** Top-down refinement uses decomposition of functions in the vocabulary and refinement of rules. Additionally, the ASM may be extended by functions and rules that are not yet considered.

**Bottom-up refinement:** Bottom-up refinement uses composition and generalisation of functions and of rules to more general or complex. Bottom-up refinement also uses generation of new functions and rules that are not yet considered.

**Modular refinement:** Modular refinement is based on parqueting of applications and separation of concern. Refinement is only applied to one module and does not affect others. Modules may also be decomposed.
*Mixed skeleton-driven refinement:* Mixed refinement is a combination of refinement techniques. It uses a skeleton of the application or a draft of the architecture. This draft is used for deriving plans for refinement. Each component or module is developed on its own based on top-down or bottom-up refinement.

These different kinds of refinement styles allow to derive *plans* for refinement and *primitives* for refinement.

### 3.3 Generic Refinement Steps and Their Correctness

An engineering approach is based on a general methodology, operations for specification evolution, and a specification of restrictions to the modelling itself. Each evolution step must either be correct according to some correctness criterion or must lead to obligations that can be used for later correction of the specification. The correctness of a refinement step is defined in terms of two given ASM together with the equivalence relations. Already [8] has observed that refinement steps can be governed by contracts. We may consider a number of governments [1]. We should however take into account the choices for style and perspectives.

![Diagram](https://via.placeholder.com/150)

**Fig. 3.** The derivation of correct refinement steps

Given a refinement pattern, perspectives, styles and contract, we may derive generic refinement steps such as data refinement, purely incremental refinement, submachine refinement, and (m,n) refinement. The generic refinement is adapted to the assumptions made for the given application and to consistency conditions. Typical such consistency are binding conditions of rules to state and vocabulary through the scope of rules. The general approach is depicted in Figure 3.
4 Conclusion

The ASM approach allows the development of a theory of database system operating. It has the following advantages:

– The signature of object-relational databases can be specified based on hierarchical predicate logic. Therefore, the theory of static integrity constraints can be entirely embedded into the theory of sequential abstract-state machines [4].

– The run of a database system is parallel. As an example that is not covered in this paper we elaborated transaction semantics that handles conflicts in concurrency. Parallel runs of transactions must behave in the same manner as a sequential run of the transactions. This concept is entirely covered by partially ordered runs.

– Interaction of database systems with user or information systems can be handled by ASM as discussed above.

– Database systems are considered on a variety of abstractions layers. Users consider a database system as an input-output engine with a powerful memory. At the conceptual layer, a database system is far more complex. The implementation is even more complex. Therefore, the consideration of database systems semantics requires also a powerful theory of refinement as provided by ASM.

Our main aim in this paper is to develop a theory of database systems at the user layer. The other abstraction layers may be seen as refinements of the business user layer. We omitted all examples to the space limitations.

The next abstraction level must consider specific approaches of database engines at the implementation layer. In this case, we need a general notion of database semantics. Database dynamics has not yet been well understood. Transactions may serve as an example. So far we have only used sequential ASM. We might however use power ASM that allow to abstract from intermediate computational steps.

Acknowledgement

The authors would like to thank Egon Börger and to Andreas Prinz for their helpful comments and discussions. Egon Börger proposed to use power ASM instead of sequential ASM for the treatment of transactions. We than P. Schmidt for proofreading.

References


Extended Entity-Relationship Model

Bernhard Thalheim
Christian-Albrechts University Kiel, http://www.informatik.uni-kiel.de/~thalheim

SYNONYMS
EERM, HERM; higher-order entity-relationship model; hierarchical entity-relationship model

DEFINITION
The extended entity-relationship (EER) model is a language for defining the structure (and functionality) of database or information systems. Its structure is developed inductively. Basic attributes are assigned to base data types. Complex attributes can be constructed by applying constructors such as tuple, list or set constructors to attributes that have already been constructed. Entity types conceptualise structuring of things of reality through attributes. Cluster types generalise types or combine types into singleton types. Relationship types associate types that have already been constructed into an association type. The types may be restricted by integrity constraints and by specification of identification of objects defined for a type. Typical integrity constraints of the extended entity-relationship model are participation, look-across, and general cardinality constraints. Entity, cluster, and relationship classes contain a finite set of objects defined on these types. The types of an EER schema are typically depicted by an EER diagram.

HISTORICAL BACKGROUND
The entity-relationship (ER) model was introduced by P.P. Chen in 1976 [1]. The model conceptualises and graphically represents the structure of the relational model. It is currently used as the main conceptual model for database and information system development. Due to its extensive usage a large number of extensions to this model were proposed in the 80’s and 90’s. Cardinality constraints [1, 3, 4, 8] are the most important generalisation of relational database constraints [7]. These proposals have been evaluated, integrated or explicitly discarded in an intensive research discussion. The semantic foundations proposed in [2, 5, 8] and the various generalisations and extensions of the entity-relationship model have led to the introduction of the higher-order or hierarchical entity-relationship model [8] which integrates most of the extensions and also supports conceptualisation of functionality, distribution [9], and interactivity [6] for information systems. Class diagrams of the UML standard are a special variant of extended entity-relationship models.

The ER conferences (annually; since 1996: International Conference on Conceptual Modeling, http://www.conceptualmodeling.org/) are the main forum for conceptual models and modelling.

SCIENTIFIC FUNDAMENTALS
The extended entity-relationship model is mainly used as a language for conceptualisation of the structure of information systems applications. Conceptualisation of database or information systems aims to represent the logical and physical structure of an information system. It should contain all the information required by the user and required for the efficient behavior of the whole information system for all users. Conceptualisation may further target the specification of database application processes and the user interaction. Structure description are currently the main use of the extended ER model.

An example of an EER diagram.
The EER model uses a formal language for schema definition and diagrams for graphical representation of the
schema. Let us consider a small university application for management of Courses. Proposed courses are based on courses and taught by a docent or an external docent within a certain semester and for a set of programs. Proposals typically include a request for a room and for a time and a categorisation of the kind of the course. Theses proposals are the basis for course planning. Planning may change time, room and kind. Planned courses are held at the university. Rooms may be changed. The example is represented by the EER diagram in Figure 1.

![Figure 1: Extended Entity-Relationship Diagram for Course Management](image_url)

Entity types are represented graphically by rectangles. Attribute types are associated with the corresponding entity or relationship type. Attributes primarily identifying a type are underlined. Relationship types are represented graphically by diamonds and associated by directed arcs to their components. A cluster type is represented by a diamond, is labelled by the disjoint union sign, and has directed arcs from the diamond to its component types. Alternatively, the disjoint union representation ⊖ is attached to the relationship type that uses the cluster type. In this case directed arcs associate the ⊖ sign with component types. An arc may be annotated with a label.

**The definition scheme for structures.**

The extended entity-relationship model uses a data type system for its attribute types. It allows the construction of entity types \( E \doteq (\text{attr}(E), \Sigma_E) \) where \( E \) is the entity type defined as a pair — the set \( \text{attr}(E) \) of attribute types and the set \( \Sigma_E \) of integrity constraints that apply to \( E \). The definition \( \text{def} \) of a type \( T \) is denoted by \( T \doteq \text{def} \).

The EER model lets users inductively build relationship types \( R \doteq (T_1, ..., T_n, attr(R), \Sigma_R) \) of order \( i \) \((i \geq 1)\) through a set of (labelled) types of order less than \( i \), a set of attribute types, and a set of integrity constraints that apply to \( R \). The types \( T_1, ..., T_n \) are the components of the relationship type. Entity types are of order 0. Relationship types are of order 1 if they have only entity types as component types. Relationship types are of order \( i \) if all component types are of order less than \( i \) and if one of the component types is of order \( i - 1 \).

Additionally, cluster types \( C \doteq T_1 \cup ... \cup T_n \) of order \( i \) can be defined through a disjoint union \( \cup \) of relationship types of order less than \( i \) or of entity types.

Entity/relationship/cluster classes \( T^C \) contain a set of objects of the entity/relationship/cluster type \( T \). The EER model mainly uses set semantics, but (multi-)list or multiset semantics can also be used. Integrity constraints apply to their type and restrict the classes. Only those classes are considered for which the constraints of their types are valid. The notions of a class and of a type are distinguished. Types describe the structure and constraints. Classes contain objects.

The data type system is typically inductively constructed on a base type \( B \) by application of constructors such as the tuple or products constructor \((..)\), set constructor \(\{..\}\), and the list constructor \(<..>\). Types may be optional component types and are denoted by \([..]\).

The types \( T \) can be labelled \( l:T \). The label is used as an alias name for the type. Labels denote roles of the type. Labels must be used if the same type is used several times as a component type in the definition of a relationship.
or cluster type. In this case they must be unique.

An entity-relationship schema consists of a set of data, attribute, entity, relationship, and cluster types which types are inductively built on the basis of the base types.

Given a base type system $B$. The types of the ER schema are defined through the type equation:

$$ T = B \mid (l_1 : T, \ldots, l_n : T) \mid \{T\} \mid <T> \mid [T] \mid T \cup T \mid l : T \mid N = T $$

Structures in detail.

The classical four-layered approach is used for inductive specification of database structures. The first layer is the data environment, called the basic data type scheme, which is defined by the system or is the assumed set of available basic data types. The second layer is the schema of a database. The third layer is the database itself representing a state of the application’s data often called micro-data. The fourth layer consists of the macro-data that are generated from the micro-data by application of view queries to the micro-data.

Attribute types and attribute values.

The classical ER model uses basic (first normal form) attributes. Complex attributes are inductively constructed by application of type constructors such as the tuple constructor (..), set constructor {..}, and the list constructor <..>. Typical base types are integers, real numbers, strings, and time. Given a set of names $\mathcal{N}$ and a set of base types $B$, a basic attribute type $A :: B$ is given by an (attribute) name $A \in \mathcal{N}$ and a base type $B$. The association between the attribute name and the underlying type is denoted by ::. The base type $B$ is often called the domain of $A$, i.e. $dom(A) = B$. Complex attributes are constructed on base attributes by application of the type constructors. The notion of a domain is extended to complex attributes, i.e. the domain of the complex attribute $A$ is given by $dom(A)$. Components of complex attributes may be optional, e.g., the Title in the attribute Name.

Typical examples of complex and basic attributes in Figure 1 are

- Name $\triangleq$ (FirstName <FirstName>, FamName, [AcadTitles], [FamilyTitle]),
- PersNo $\triangleq$ EmplNo $\cup$ SocSecNo,
- AcadTitles $\triangleq$ [AcadTitle],
- Contact $\triangleq$ (Phone({PhoneAtWork}, private), Email, URL, WebContact, [Fax({PhoneAtWork})]) ,
- PostalAddress $\triangleq$ (Zip, City, Street, HouseNumber)

for DateOfBirth :: date, AcadTitle :: acadTitleType, FamilyTitle :: familyTitleAcronym, Zip :: string7,

- SocSecNo :: string9, EmplNo :: int, City :: varString, Street :: varString, HouseNumber :: smallInt.

The complex attribute Name is structured into a sequence of first names, a family name, an optional complex set-valued attribute for academic titles, and an optional basic attribute for family titles. Academic titles and family titles can be distinguished from each other.

Entity types and entity classes.

Entity types are characterized by their attributes and their integrity constraints. Entity types have a subset $K$ of the set of attributes which serve to identify the objects of the class of the type. This concept is similar to the concept of key known for relational databases. The key is denoted by ID(K). The set of integrity constraints $\Sigma_E$ consists of the keys and other integrity constraints. Identifying attributes may be underlined instead of having explicit specification.

Formally, an entity type is given by a name $E$, a set of attributes $attr(E)$, a subset $id(E)$ of $attr(E)$, and a set $\Sigma_E$ of integrity constraints, i.e.

$$ E \triangleq \langle attr(E), \Sigma_E \rangle. $$

The following types are examples of entity types in Figure 1:

- Person $\triangleq$ ( { Name, Login, URL, Address, Contact, DateOfBirth, PersNo } )
- Course $\triangleq$ ( { CourseID, Title, URL }, { ID( { CourseID } ) } ),
- Room $\triangleq$ ( { Building, Number, Capacity }, { ID({ Building, Number } }) ),
- Semester $\triangleq$ ( { Term, Date(Starts, Ends) }, { ID({ Term } ) } )

An ER schema may use the same attribute name with different entity types. For instance, the attribute URL in Figure 1 is used for characterising additional information for the type Person and the type Course. If they need to be distinguished, then complex names such as CourseURL and PersonURL are used.

Objects on type $E$ are tuples with the components specified by a type. For instance, the object (or entity) (HRS3, 408A, 15) represents data for the Room entity type in Figure 1.
An entity class $E_C$ of type $E$ consists of a finite set of objects on type $E$ for which the set $\Sigma_E$ of integrity constraints is valid.

Cluster types and cluster classes.
A disjoint union $\cup$ of types whose identification type is domain compatible is called a cluster. Types are domain compatible if they are subtypes of a common more general type. The union operation is restricted to disjoint unions since identification must be preserved. Otherwise, objects in a cluster class cannot be related to the component classes of the cluster type. Cluster types can be considered as a generalisation of their component types.

A cluster type (or “category”)

\[ C \equiv l_1 : R_1 \cup l_2 : R_2 \cup \ldots \cup l_k : R_k \]

is the (labelled) disjoint union of types $R_1, \ldots, R_k$. Labels can be omitted if the types can be distinguished.

The following type is an example of a cluster type:

\[ \text{Teacher} \equiv \text{ExternalDocent} : \text{CollaborationPartner} \cup \text{Docent} : \text{Professor} \]

The cluster class $C_C$ is the ‘disjoint’ union of the sets $R_1^C, \ldots, R_k^C$. It is defined if $R_1^C, \ldots, R_k^C$ are disjoint on their identification components. If the sets $R_1^C, \ldots, R_k^C$ are not disjoint then labels are used for differentiating the objects of clusters. In this case, an object uses a pair representation $(l_i, o_i)$ for objects $o_i$ from $R_i^C$.

Relationship types and relationship classes.
First order relationship types are defined as associations between entity types or clusters of entity types. Relationship types and clusters are also defined on the basis of relationship types that are already defined. This construction must be inductive and cannot be cyclic. Therefore, an order is introduced for relationship types. Types can only be defined on the basis of types which have a lower order. For instance, the type Professor in Figure 1 is of order 1. The type ProposedCourse is of order 2 since all its component types are either entity types or types of order 1. A relationship type of order $i$ is defined as an association of relationship types of order less than $i$ or of entity types. It is additionally required that at least one of the component types is of order $i - 1$ if $i > 1$. Relationship types can also be characterized by attributes. Relationship types with one component type express a subtype or an Is-A relationship type. For instance, the type Professor is a subtype of the type Person.

Component types of a relationship type may be labelled. Label names typically provide an understanding of the role of a component type in the relationship type. Labelling uses the definition scheme $Label : Type$. For instance, the Kind entity type is labelled by Proposal for the relationship type ProposedCourse in in Figure 1. Cluster types have the maximal order of their component types. Relationship types also may have cluster type components. The order of cluster type components of a relationship type of order $i$ must be less than $i$.

Component types that are not used for identification within the relationship type can be optional. For example, the Kind component for a PlannedCourse is optional for the type ProposedCourse. If the relationship object in the PlannedCourse class does not have a room then the proposal for rooms in ProposedCourse is accepted. A specific extension for translation of optional component types may be used. For instance, Room in Figure 1 is inherited to PlannedCourse from ProposedCourse if the Room component for a PlannedCourse is missing.

Higher order types allow a convenient description of types that are based on other types. For example, consider the course planning application in Figure 1. Lectures are courses given by a professor or a collaboration partner within a semester for a number of programs. Proposed courses extend lectures by describing which room is requested and which time proposals and which restrictions are made. Planning of courses assigns a room to a course that has been proposed and assigns a time frame for scheduling. The kind of the course may be changed. Courses that are held are based on courses planned. The room may be changed for a course. The following types specify these assertions.

\[ \text{ProposedCourse} \equiv \{ \text{Teacher}, \text{Course}, \text{Proposal} : \text{Kind}, \text{Request} : \text{Room}, \text{Semester}, \text{Set2} : \{ \text{Program} \}, \{ \text{Time(Proposal, SideCondition)} \}, \Sigma_{\text{ProposedCourse}} \} \]

\[ \text{PlannedCourse} \equiv \{ \text{ProposedCourse}, [\text{Reassigned} : \text{Kind}], [\text{Reassigned} : \text{Room}], \{ \text{TimeFrame, TermCourseID} \}, \Sigma_{\text{PlannedCourse}} \} \]

\[ \text{CourseHeld} \equiv \{ \text{PlannedCourse}, [\text{Reassigned} : \text{Room}], \{ \text{StartDate, EndDate, AssistedBy} \}, \Sigma_{\text{CourseHeld}} \} \]

The second and third types use optional components in case a proposal or a planning of rooms or kinds is changed. Typically, planned courses are identified by their own term-specific identification. Integrity constraints can be
Formally, a relationship type is given by a name $R$, a set $\text{compon}(R)$ of labelled components, a set of attributes $\text{attr}(R)$, and a set $\Sigma_R$ of integrity constraints that includes the identification of the relationship type by a subset $\text{id}(R)$ of $\text{compon}(R) \cup \text{attr}(R)$, i.e.

$$R \triangleq (\text{compon}(R), \text{attr}(R), \Sigma_R) .$$

It is often assumed that the identification of relationship types is defined exclusively through their component types. Relationship types that have only one component type are unary types. These relationship types define subtypes. If subtypes need to be explicitly represented then binary relationship types named by $\text{IsA}$ between the subtype and the supertype are used. For instance, the type $\text{Professor}$ in Figure 1 is a subtype of the type $\text{Person}$. An object (or a “relationship”) on the relationship type $R \triangleq (R_1, \ldots, R_n, \{B_1, \ldots, B_k\})$, $\text{id}(R)$, $\Sigma_R$) is an element of the Cartesian product $R'_1 \times \ldots \times R'_n \times \text{dom}(B_1) \times \ldots \times \text{dom}(B_k)$. A relationship class $R^C$ consists of a finite set $R^C \subseteq R'_1 \times \ldots \times R'_n \times \text{dom}(B_1) \times \ldots \times \text{dom}(B_k)$ of objects on $R$ for which $\text{id}(R)$ is a key of $R^C$ and which obeys the constraints $\Sigma_R$.

### Integrity constraints.

Each database model also uses a set of implicit model-inherent integrity constraints. For instance, relationship types are defined over their component types, and a (relationship) object presumes the existence of corresponding component objects. Typically only finite classes are considered. The EER schema is acyclic. Often names or labels are associated with a minimal semantics that can be derived from the meaning of the words used for names or labels. This minimal semantics allows us to derive synonym, homonym, antonym, troponym, hypernym, and holonym associations among the constructs used.

The most important class of integrity constraints of the EER model is the class of cardinality constraints. Other classes of importance for the EER model are multivalued dependencies, inclusion and exclusion constraints and existence dependencies[7]. Functional dependencies, keys and referential constraints (or key-based inclusion dependencies) can be expressed through cardinality constraints.

Three main kinds of cardinality constraints are distinguished: participation constraints, look-across constraints, and general cardinality constraints. Given a relationship type $R \triangleq (\text{compon}(R), \text{attr}(R), \Sigma_R)$, a component $R'$ of $R$, the remaining substructure $R'' = R \setminus R'$ and the remaining substructure $R''' = R'' \cap_R \text{compon}(R)$ without attributes of $R$.

The participation constraint $\text{card}(R, R') = (m, n)$ restricts the number of occurrences of $R'$ objects in the relationship class $R^C$ by the lower bound $m$ and the upper bound $n$. It holds in a relationship class $R^C$ if for any object $o' \in R^C$ there are at least $m$ and at most $n$ objects $o \in R^C$ with $\pi_{R'}(o) = o'$ for the projection function $\pi_{R'}$ that projects $o$ to its $R'$ components.

Participation constraints relate objects of relationship classes to objects of their component classes. For instance, the constraint $\text{card}(\text{ProposedCourse}, \text{SemesterCourse}) = (0, 3)$ restricts relationship classes for proposals for courses per semester to at least 0 and at most 3, i.e. each course is proposed at most three times in a semester. There are at most three objects $o$ in $\text{ProposedCourse}^C$ with the same course and semester objects. The integrity constraint $\text{card}(\text{ProposedCourse}, \text{DocentSemester}) = (3, 7)$ requires that each docent is giving at least 3 courses and at most 7 courses. External docents may be obliged by other restrictions, e.g., $\text{card}(\text{ProposedCourse}, \text{ExternalDocentSemester}) = (0, 1)$.

Formally, the integrity constraint $\text{card}(R, R') = (m, n)$ is valid in $R^C$ if $m \leq \left| \{ o \in R^C : \pi_{R'}(o) = o' \} \right| \leq n$ for any $o' \in \pi_{R'}(R^C)$ and the projection $\pi_{R'}(R^C)$ of $R^C$ to $R'$.

If $\text{card}(R, R') = (0, 1)$ then $R'$ forms an identification or a key of $R$, i.e. $\text{ID}(R')$ for $R$. This identification can also be expressed by a functional dependency $R : R' \rightarrow R''$.

The lookup or look-across constraint $\text{look}(R, R') = m..n$ describes how many objects $o''$ from $R'''$ may potentially ‘see’ an object $o'$ from $R^C$. It holds in a relationship class $R^C$ if for any object $o'' \in \text{dom}(R''')$ there are at least $m$ and at most $n$ related objects $o'$ with $\pi_{R'}(o) = o'$, i.e. $m \leq \left| \{ o' \in \pi_{R'}(R^C) : o \in R^C \land \pi_{R'}(o) = o' \land \pi_{R'''}(o) = o'' \} \right| \leq n$ for any $o'' \in \text{dom}(R''')$. Typically, look-across constraints are used for components consisting of one type. Look-across constraints are not defined for relationship types with one component type.

Look-across constraints are less intuitive for relationship types with more than 2 component types or with attribute types. For instance, the look-across constraint $\text{look}(\text{ProposedCourse}, \text{DocentSemester}) = 0..7$ specifies that for any combination of $\text{Teacher}$, $\text{Room}$, $\text{Kind}$, and $\text{Program}$ objects there are between 0 and 7 $\text{Docent}$ and $\text{Semester}$
dependency. The types The first constraint does not restrict the database. The second constraint expresses a key or functional

An EER schema is defined by the pair as components and cluster and relationship types are properly layered. A set 

The schema is based on a set of base (data) types which are used as value types for attribute types. Logical operators can be defined for each type. A set of logical formulas using these operators can define the functional dependencies and referential integrity constraints into a singleton construct.

Look-across constraints were originally introduced by P.P. Chen [1] as cardinality constraints. UML uses look-across constraints. Participation and look-across constraints cannot be axiomatised through a Hilbert- or Gentzen-type logical calculus. If only upper bounds are of interest then an axiomatisation can be found in [3] and [4]. General cardinality constraints combine equality-generating and object-generating constraints such as keys, functional dependencies and referential integrity constraints into a singleton construct. Logical operators can be defined for each type. A set of logical formulas using these operators can define the integrity constraints which are valid for each object of the type.

Schemata.
The schema is based on a set of base (data) types which are used as value types for attribute types. A set \( \{E_1, ..., E_n, C_1, ..., C_l, R_1, ..., R_m\} \) of entity, cluster and (higher-order) relationship types on a data scheme DD is called schema if the relationship and cluster types use only the types from \( \{E_1, ..., E_n, C_1, ..., C_l, R_1, ..., R_m\} \) as components and cluster and relationship types are properly layered. An EER schema is defined by the pair \( \mathcal{D} = (S, \Sigma) \) where \( S \) is a schema and \( \Sigma \) is a set of constraints. A database \( \mathcal{D}^C \) on \( \mathcal{D} \) consists of classes for each type in \( \mathcal{D} \) such that the constraints \( \Sigma \) are valid. The classes of the extended ER model have been defined through sets of objects on the types. In addition to sets, lists, multi-sets or other collections of objects may be used. In this case, the definitions used above can easily be extended [8].

A number of domain-specific extensions have been introduced to the ER model. One of the most important is the extension of the base types by spatial data types such as: point, line, oriented line, surface, complex surface, oriented surface, line bunch, and surface bunch. These types are supported by a large variety of functions such as: meets, intersects, overlaps, contains, adjacent, planar operations, and a variety of equality predicates. The translation of the schema to (object-)relational or XML schemata can be based on a profile [8]. Profiles define which translation choice is preferred over other choices, how hierarchies are treated, which redundancy and null-value support must be provided, which kind of constraint enforcement is preferred, which naming conventions are chosen, which alternative for representation of complex attributes is preferred for which types, and whether weak types can be used. The treatment of optional components is also specified through the translation profile of the types of the schema. A profile may require the introduction of identifier types and base the identification on the identifier. Attribute types may be translated into data formats that are supported by the target system.
The EER schema can be used to define views. The generic functions insert, delete, update, projection, union, join, selection and renaming can be defined in a way similarly to the relational model. Additionally, nesting and unnesting functions are used. These functions form the algebra of functions of the schema and are the basis for defining queries. A singleton view is defined by a query that maps the EER schema to new types. Combined views also may be considered which consist of singleton views which together form another EER schema.

A view schema is specified over an EER schema \( D \) by a schema \( V = \{ S_1, ..., S_m \} \), an auxiliary schema \( A \) and a (complex) query \( q : D \times A \rightarrow V \) defined on \( D \) and \( A \). Given a database \( D^C \) and the auxiliary database \( A^C \). The view is defined by \( q(D^C \times A^C) \).

**Graphical representation.**

The schema in Figure 1 consists of entity, cluster and relationship types. The style of drawing diagrams is one of many variants that have been considered in the literature. The main difference of representation is the style of drawing unary types. Unary relationship types are often represented by rectangles with rounded corners or by (directed) binary IsA-relationship types which associate by arcs the supertype with the subtype. Tools often do not allow cluster types and relationship types of order higher than 1. In this case, those types can be objectified, i.e. represented by a new (abstract) entity type that is associated through binary relationship types to the components of the original type. In this case, identification of objects of the new type is either inherited from the component types or is provided through a new (surrogate) attribute. The first option results in the introduction of so-called weak types. The direct translation of these weak types to object-relational models must be combined with the introduction of rather complex constraint sets. Typically, this complexity can be avoided if the abstract entity type is mapped together with the new relationship types to a singleton object-relational type. This singleton type is also the result of a direct mapping of the original higher-order relationship type.

The diagram can be enhanced by an explicit representation of cardinality and other constraints. If participation constraints \( \text{card}(R, R') = (m, n) \) are used for component consisting of one type \( R' \) then the arc from \( R \) to \( R' \) is labelled by \( (m, n) \). If look-across constraints \( \text{look}(R, R') = m \ldots n \) are used for binary relationship types then the arc from \( R \) to \( R' \) is labelled by \( m \ldots n \).

**KEY APPLICATIONS**

The main application area for extended ER models is the conceptualisation of database applications. Database schemata can be translated to relational, XML or other schemata based on transformation profiles that incorporate properties of the target systems.

**FUTURE DIRECTIONS**

The ER model has had a deep impact on the development of diagramming techniques in the past and is still influencing extensions of the unified modelling language UML. UML started with binary relationship types with look-across constraints and without relationship type attributes. Class diagrams currently allow n-ary relationship types with attributes. Relationship types may be layered. Cluster types and unary relationship types allow for distinguishing generalisation from specialisation.

ER models are not supported by native database management systems and are mainly used for modelling of applications at the conceptual or requirements level. ER schemata are translated to logical models such as XML schemata or relational schemata or object-relational schemata. Some of the specifics of the target models are not well supported by ER models and must be added after translating ER schemata to target schemata, e.g., specific type semantics such as list semantics (XML) or as special ordering or aggregation treatment of online analytical processing (OLAP) applications.

The ER model has attracted a lot of research over the last 30 years. Due to novel applications and to evolution of technology old problems and novel problems are challenging the research on this model. Typical old problems that are still not solved in a satisfactory manner are: development of a science of modelling, quality of ER schemata, consistent refinement of schemata, complex constraints, normalisation of ER schemata, normalisation of schemata in the presence of incomplete constraint sets. Novel topics for ER research are for instance: evolving schema architectures, collaboration of databases based on collaboration schemata, layered information systems...
and their structuring, schemata with redundant types, ER schemata for OLAP applications. Structures of database applications are often represented through ER models. Due to the complexity of applications, a large number of extensions have recently been proposed, e.g., temporal data types, spatial data types, OLAP types and stream types. Additionally, database applications must be integrated and cooperate in a consistent form. The harmonisation of extensions and the integration of schemata is therefore a never ending task for database research.

ER models are currently extended for support of (web) content management that is based on structuring of data, on aggregation of data, on extending data by concepts and on annotating data sets for simple reference and usage. These applications require novel modelling facilities and separation of syntactical, semantical and pragmatic issues. The ER model can be extended to cope with these applications. The ER model is mainly used for conceptual specification of database structuring. It can be enhanced by operations and a query algebra. Operations and the queries can also be displayed in a graphical form, e.g., on the basis of VisualSQL. Most tools supporting ER models do not currently use this option. Enhancement of ER models by functionality is necessary if the conceptualisation is used for database development. Based on functionality enhancement, view management facilities can easily be incorporated into these tools.

ER models are becoming a basis for workflow systems data. The standards that have been developed for the specification of workflows have not yet been integrated into sophisticated data and application management tools.

URL TO CODE
http://www.informatik.uni-kiel.de/~thalheim/HERM.htm
http://www.is.informatik.uni-kiel.de/~thalheim/indeeerm.htm

Readings on the RADD project (Rapid Application and Database Development)

CROSS REFERENCE
I. DATABASE FUNDAMENTALS
   a. Data models (including semantic data models)
   b. Entity-Relationship (ER) model
   c. Unified modelling language (UML)

III. THEORETICAL ASPECTS
   b. Relational Theory

RECOMMENDED READING
Between 3 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.

Specialisation and Generalisation

Bernhard Thalheim

Christian-Albrechts University Kiel, http://www.informatik.uni-kiel.de/~thalheim/HERM.htm

SYNONYMS
refinement, abstraction, hierarchies;
clustering, grouping, inheritance

DEFINITION
Generalisation and specialisation are main principles of database modelling. Generalisation maps or groups types or classes to more abstract or combined ones. It is used to combine common features, attributes, or methods. Specialisation is based on a refinement of types or classes to more specific ones. It allows developers to avoid null values and to hide details from non-authorised users. Typically, generalisations and specialisations form a hierarchy of types and classes. The more specialised classes may inherit attributes and methods from more general ones. In database modelling and implementation clusters of types to a type that represents common properties and abstractions from a type are the main kinds of generalisations. Is-A associations that specialise a type to a more specific one and Is-A-Role-Of associations that considers a specific behaviour of objects are the main kinds of specialisations.

MAIN TEXT
Specialisation introduces a new entity type by adding specific properties belonging to that type which are different from the general properties of its more general type. Generalisation introduces the Role-Of relationship or the Is-A relationship between a subtype and its general type. Therefore, the application, implementation, and processes are different. For generalisation the general type must be the union of its subtypes. The subtypes can be virtually clustered by the general type. This tends not to be the case for specialisation. Specialisation is a refinement or restriction of a type to more special ones. Typical specialisations are Is-A and Has-Role associations. Exceptions can be modelled by specialisations.

Different kinds of specialisation may be distinguished: structural specialisation which extends the structure, semantic specialisation which strengthens type restrictions, pragmatical specialisation which allows to separate the different usage of objects in contexts, operational specialisation which introduces additional operations, and hybrid specialisations. Is-A specialisation requires structural and strong semantic specialisation. Is-A-Role-Of specialisation requires structural, pragmatical and strong semantic specialisation.

Generalisation is based either on abstraction or on grouping. The cluster construct of the extended ER model is used to represent generalisations. Generalisation tends to be an abstraction in which a more general type is defined by extracting common properties of one or more types while suppressing the differences between them. These types are subtypes of the generic type. New types are created by generalizing classes that already exist. Structural combination typically assumes the existence of a unifiable identification of all types. Semantical combination allows the disjunction of types through the linear sum of semantics. Pragmatical generalisation is based on building collections whenever applications require a consideration of commonalties.

CROSS REFERENCE
I. DATABASE FUNDAMENTALS
a. Data models (including semantic data models)

REFERENCES
Abstraction

Bernhard Thalheim

Christian-Albrechts University Kiel, http://www.informatik.uni-kiel.de/˜thalheim/HERM.htm

SYNONYMS
- component abstraction, localisation abstraction, implementation abstraction;
- association, aggregation, composition, grouping, specialisation, generalisation, classification

DEFINITION
Abstraction allows developers to concentrate on the essential, relevant or important parts of an application. It uses a mapping to a model from things in reality or from virtual things. The model has the truncation property, i.e. it lacks some of the details in the original, and a pragmatic property, i.e. the model use is only justified for particular model users, tools of investigation, and periods of time. Database engineering uses construction abstraction, context abstraction and refinement abstraction. Construction abstraction is based on the principles of hierarchical structuring, constructor composition, and generalisation. Construction abstraction assumes that the surroundings of a concept are commonly assumed by a community or within a culture and focuses on the concept, turning away attention from its surroundings such as the environment and setting. Refinement abstraction uses the principle of modularisation and information hiding. Developers typically use conceptual models or languages for representing and conceptualising abstractions. The enhanced entity-relationship model schema are typically depicted by an EER diagram.

MAIN TEXT
Database engineering distinguishes three kinds of abstraction: construction abstraction, context abstraction and refinement abstraction.
Constructor composition depends on the constructors as originally introduced by J. M. Smith and D.C.W. Smith. Composition constructors must be well founded and their semantics must be derivable by inductive construction. There are three main methods for construction: development of ordered structures on the basis of hierarchies, construction by combination or association, and construction by classification into groups or collections. The set constructors \( \subseteq \) (subset), \( \times \) (product) and \( \mathcal{P} \) (powerset) for subset, product and nesting are complete for the construction of sets.
Subset constructors support hierarchies of object sets in which one set of objects is a subset of some other set of objects. Subset hierarchies are usually a rooted tree. Product constructors support associations between object sets. The schema is decomposed into object sets related to each other by association or relationship types. Power set constructors support a classification of object sets into clusters or groups of sets - typically according to their properties.

Context abstraction allows developers to commonly concentrate on those parts of an application that are essential for some viewpoints during development and deployment of systems. Typical kinds of context abstraction are component abstraction, separation of concern, interaction abstraction, summarisation, scoping, and focusing on typical application cases.
Component abstraction factors out repeating, shared or local patterns of components or functions from individual concepts. It allows developers to concentrate on structural or behavioral aspects of similar elements of components. Separation of concern allows developers to concentrate on those concepts that are a matter of development and to neglect all other concepts that are stable or not under consideration. Interaction abstraction allows developers to concentrate on those parts of the model that are essential for interaction with other systems or users. Summarisation maps the conceptualisations within the scope to more abstract concepts. Scoping is typically used to select those concepts that are necessary for current development and removes those concepts that do not have an impact on the necessary concepts.
Database models may cover a large variety of different application cases. Some of them reflect exceptional,
abnormal, infrequent and untypical application situations. Focusing on typical application cases explicitly separates models for the normal or typical application case from those that are atypical. Atypical application cases are not neglected but can be folded into the model whenever atypical situations are considered. The context abstraction concept is the main concept behind federated databases. Context of databases can be characterized by schemata, version, time, and security requirements. Sub-schemata, types of the schemata or views on the schemata, are associated by explicit import/export bindings based on a name space. Parametrisation lets developers to consider collections of objects. Objects are identifiable under certain assumptions and completely identifiable after instantiation of all parameters. Interaction abstraction allows developers to display the same set of objects in different forms. The view concept supports this visibility concept. Data is abstracted and displayed in various levels of granularity. Summarisation abstraction allows developers to abstract from details that are irrelevant at a certain step. Scope abstraction allows developers to concentrate on a number of aspects. Names or aliases can be multiply used with varying structure, functionality and semantics.

Refinement abstraction is mainly about implementation and modularisation. It allows developers to selectively retain information about structures. Refinement abstraction is defined on the basis of the development cycle (refinement of implementations). It refines, summarises and views conceptualizations, hides or encapsulates details or manages collections of versions. Each refinement step transforms a schema to a schema of finer granularity. Refinement abstraction may be modelled by refinement theory and infomorphisms. Encapsulation removes internal aspects and concentrates on interface components. Blackbox or graybox approaches hide all aspects of the objects under consideration. Partial visibility may be supported by modularisation concepts. Hiding supports differentiation of concepts into public, private (with the possibility to be visible to ‘friends’) and protected (with visibility to subconcepts). It is possible to define a number of visibility conceptualizations based in inflection. Inflection is used for the injection of combinable views into the given view, for tailoring, ordering and restructuring of views, and for enhancement of views by database functionality. Behavioral transparency is supported by the glassbox approach. Security views are based on hiding. Versioning allows developers to manage a number of concepts which can be considered to be versions of each other.

CROSS REFERENCE
I. DATABASE FUNDAMENTALS
   a. Entity-Relationship Model, Extended Entity-Relationship Model, Object Data Models,
      Object Role Modeling, Unified Modeling Language

REFERENCES