

Horizontal Business Process Model Integration

Klaus-Dieter Schewe^{1,2}✉, Verena Geist¹, Christa Illibauer¹, Felix Kossak¹,
Christine Natschläger-Carpella¹, Theodorich Kopetzky¹, Jan Kubovy²,
Bernhard Freudenthaler¹, and Thomas Ziebermayr¹

¹ Software Competence Center Hagenberg, Hagenberg im Mühlkreis, Austria
{kd.schewe, verena.geist, christa.illibauer, felix.kossak,
christine.natschlaeger, theodorich.kopetzky, bernhard.freudenthaler,
thomas.ziebermayr}@scch.at

² Johannes-Kepler-University Linz, Linz, Austria
{kd.schewe, jkubovy}@faw.jku.at

Abstract. Modelling business processes in general is a complex endeavour, as many different aspects such as the control flow, the management of data, event and message handling, actors and interaction, exception handling, etc. have to be taken into account, all of which require different models. This paper focuses on the horizontal integration of models for control flow, message flow, event handling, interaction, actors, data and exception handling. The method is based on Abstract State Machines (ASMs), which are used to formally define the semantics of each of the individual models. Throughout the process rigorous quality assurance methods will be applied.

1 Introduction

Modelling information systems in general is a complex endeavour, as systems comprise many different aspects such as the data, functionality, interaction, distribution, context, etc., which all require different models. In addition, models are usually built on different levels of abstraction and the switch from one of these levels to another one may cause mismatches. Horizontal model integration refers to the creation of system models by successive enlargement, whereas vertical model integration refers to the systematic, seamless refinement process of high-level abstract (conceptual) models down to running systems.

In this paper we concentrate on the horizontal integration of business process models following the approach sketched in [27]. Taking a meromorphic view we consider complex systems as aggregations of parts, each requiring a different model. Then the extension of one submodel by another one is formally handled by refinement capturing interfaces and overlaps in a consistent way. Key questions to be addressed concern the provision of a clear semantics for the integration, the

The research reported in this paper was supported by the European Fund for Regional Development as well as the State of Upper Austria for the project *Vertical Model Integration* within the program “Regionale Wettbewerbsfähigkeit Oberösterreich 2007–2013”.

understanding of the information capacity of integrated models through notions of dominance and equivalence, and the integration into the process of requirements elicitation, refinement, validation and verification.

Business process modelling has a long tradition in research with many modelling approaches such as BPMN [30], YAWL [29], ARIS [26] or S-BPM [11], just to mention a few¹. Syntactical errors can be checked with the help of an ontology defining the concepts in BPMN [18]. However, a key concern is that though all methods claim to have reached a high level of maturity, semantics [1, 10, 32], flexibility [25] and adequacy for the problem domain [31, 33] are still matters of concern. Surprisingly, still many relevant aspects of business processes are not well covered, e.g. data handling or exception handling, while others are overloaded. In other words, the issue of semantics is still open as discussed in detail by Börger in [4]. Our own research started from BPMN and so far closed several semantic gaps, which will be reported in the monograph [16].

1.1 Our Approach

With respect to horizontal model integration it is common to start with the *control flow model*, i.e. a business process is described in an abstract way by a set of activities and gateways, the latter ones for splitting and synchronisation, plus start and termination events. Depending on whether one, all or an arbitrary selection of (outgoing) paths are enabled in splitting gateways, we adopt the common distinction between XOR-, AND- and OR-gateways with an analogous distinction for the synchronisation gateways. However, this terminology is in a sense misleading, as there need not be a well-nested structure, in which a splitting-gateway corresponds to exactly one synchronisation gateway. This is one of the reasons, why we formalise the semantics of each of the constructs by means of Abstract State Machines (ASMs, [7]). As a state-based rigorous method, ASMs support the unambiguous capture of the semantics [5, 8], in particular for OR-synchronisation [6]. Furthermore, on grounds of ASMs necessary subtle distinctions and extensions to the control flow model such as counters, priorities, freezing, etc. can be easily integrated in a smooth way. All constructs found in a control flow model are supposed to be executed in parallel for all process instances.

The control flow model is then extended by a *message model* and an *event model*. For this refinement in ASMs – mainly conservative extensions – are exploited [15]. In particular, the ground specification of firing conditions that depend on the state of the control flow, data, events and resources and actions that update this state [9] requires that only conditions and actions are refined. While messages are easily captured by means of specifications of sender and receiver, it becomes more subtle to define details such as synchronised vs. asynchronised messaging, delivery failure, rejection, message box overflows, etc. In our approach the ASM-based specification of messaging from S-BPM [11] has been adopted. For the event model it is necessary and sufficient to specify

¹ The survey in [24] tries to give a comparative evaluation.

what kind of events are to be observed, which can be captured on the grounds of monitored locations in ASMs, and which event conditions are to be integrated into the model.

The next horizontal extensions concern the *actor model*, i.e. the specification of responsibilities for the execution of activities (roles), as well as rules governing rights and obligations. This leads to the integration of deontic constraints [17, 22], some of which can be exploited to simplify the control flow [20, 21] or to handle optionality [19]. In this way subtle distinctions regarding decision-making responsibilities in BPM can be captured. Horizontal model integration through refinement is then extended towards an *interaction model* and a *data model*. For this, an abstract dialogue model is adopted (see [28] or similarly [12, 14]) capturing interaction by means of operations on views that are defined on top of a database schema. In this way the data model results from view integration, but global consistency has to be addressed, as a global database infers dependencies between activities that are not visible on the control flow level.

Finally, an *exception handling model* has to be integrated to complete the horizontal integration picture. This is still in a preliminary state in our work. Overall, the general idea is that an exception is a disruptive event that requires partial rollback and depending on the state the continuation with a different subprocess.

1.2 Outline of This Article

In Sect. 2 we discuss simple control flow specifications. While syntactically we stay close to the BPMN approach with respect to activities and gateways we define semantics on grounds of ASMs and discuss some fundamental problems. We also show simple verification examples. In Sect. 3 we continue this discussion of control flow focussing now on extensions by flags and counters and their use. We also extend the discussion on verification. Then Sect. 4 is dedicated to messages and events, by means of which the control flow will be enriched by additional conditions and actions. This is followed by a discussion of data handling in Sect. 5, where we stress the importance of a two-layered approach separating the external data handling by means of dialogues and views supporting the activities in the control flow and the internal data handling by means of underlying databases, the associated problems of transaction handling and view updates, and the impact on the semantics of the business processes. This is taken further in Sect. 6 addressing actors, roles and associated deontic concepts of permission and obligation. In this context we also briefly discuss principles of exception handling. We conclude in Sect. 7 with a brief summary and a discussion of further extensions needed with respect to horizontal business model integration. We also emphasise the role of complementary vertical business process model integration, which is outside the scope of this article.

2 Simple Control Flows

In this section we discuss the semantics and verification of simple control flow specifications as they appear in most control-oriented BPM specification methods, e.g. in BPMN. Our presentation stays close to BPMN, but we question some semantic declarations and require concretisations.

2.1 Components of Simple Control Flows

Basically, a **control flow specification** is a directed graph specifying the sequencing of (basic, i.e. atomic) activities. To be able to illustrate our very fundamental approach to semantics, let us for the moment restrict ourselves to only the following core components that are permitted in a simple control flow:

- **Activities:** An activity marks that someone has to execute a particular task such as compiling an order, writing a review, endorsing an application, preparing a delivery, composing a bill, etc. We consider activities to be atomic, but nonetheless the execution may need some time, and for the time being the duration of the execution is not specified. We only assume that eventually an activity that has been activated will also be terminated.
- **Start and End events:** A start event simply marks that a process will be initiated here. There is only one edge going out of a start event. A start event marks the termination of the process. There may be more than one edge leading to an end event.
- **Gateways:** A gateway enables the specification that the control flow is *split* into several branches, or analogously several branches are *synchronised* in a single continuing flow.

The simplest split gateways are *exclusive* and *parallel splits*. In the former case exactly one of the continuation paths will be followed, in the latter case all continuation paths will be followed. It is not foreseen to restrict gateways in a way that for each split there is a corresponding matching join. It may well be the case that a split is followed by other splits, where some branches will be synchronised, but it is also possible that no synchronisation is specified at all².

Analogously, there are *exclusive* and *parallel* synchronisation gateways, also denoted as *join* gateways. In the former case a single incoming flow will be passed on – we will have to discuss this later in this section. In the latter case all incoming flows must have been completed in order to continue with a single flow.

2.2 Examples

Let us look at some examples illustrated by Figs. 1, 2, 3 and 4. Though the core components in control flows are similar to those used by BPMN, we slightly

² Therefore, the notion of XOR-split and AND-split used as synonyms in the literature are misleading, as in general there is no well-defined bracket structure.

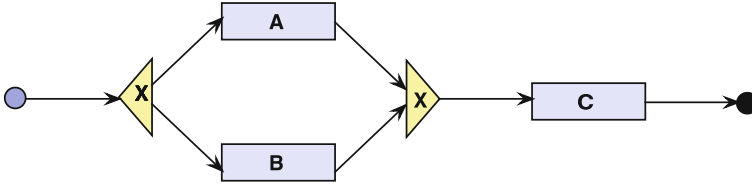


Fig. 1. Control flow with exclusive split and join

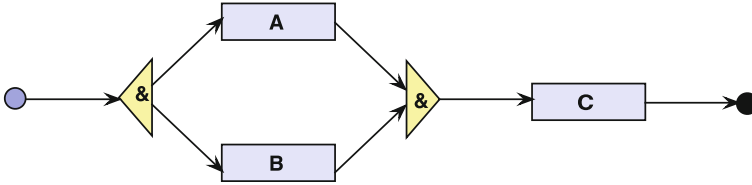


Fig. 2. Control flow with parallel split and join

modify the graph notation. In particular, we want to highlight the difference between split-gateways (one incoming edge and several outgoing ones) and synchronisation gateways (several incoming edges and only one outgoing one), so we use triangles instead of diamonds. In this way, we can reserve diamonds for gateways with several incoming and several outgoing flows, i.e. complex gateways. Furthermore, we use labels **X** and **&** for exclusive and parallel (split and join) gateways, respectively.

Example 1. Figure 1 shows a control flow with an exclusive split and a matching join. That is, after start the process will be either continued by activity **A** or activity **B**, followed by activity **C**. Similarly, Fig. 2 shows a control flow with a parallel split and a matching join. That is, after start the process will be continued by executing activities **A** and **B** in parallel, followed by activity **C**. So in both cases the informal semantics of the control flow is clear.

Example 2. This is not so clear for the control flow in Fig. 3, as the parallel split is matched by an exclusive join. Informally, this means that after start both activities **A** and **B** are executed in parallel, but what is the meaning of the synchronising exclusive join? If control is simply passed on as foreseen in BPMN, then activity **C** will be executed twice. If, however, this is not desired, the control flow could be considered to be incorrect, in which case this should be detected, and the control flow should not be permitted. Alternatively, it could still be considered to be correct, if the exclusive join gateway only passes the control on after the first completion of either **A** or **B**, whereas any follow-on enabling of the gateway would be ignored. In this case, however, the semantics of an exclusive join has to be specified in a way that permits to keep track, if still some information may arrive or not.

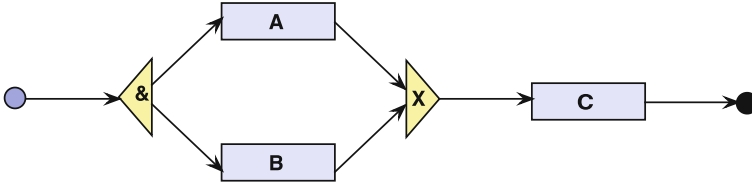


Fig. 3. Control flow with parallel split and exclusive join

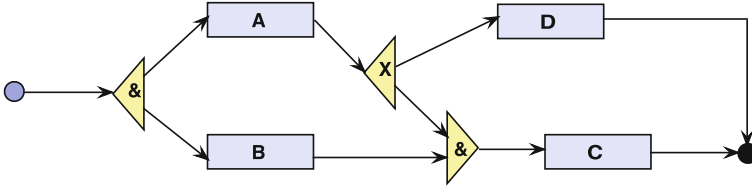


Fig. 4. Control flow without matching split and join gateways

Example 3. Finally, look at Fig. 4. In this case it seems informally clear that the control flow specification is not correct. After start activities **A** and **B** would be executed in parallel. After completion of **A** an exclusive split would either enable activity **D** or pass the control onto the parallel join gateway following activity **B**. That is, if **D** is enabled, this parallel join will wait forever, and activity **C** cannot be enabled. Only if control is not passed onto **D**, the parallel join would (after completion of **B**) pass control onto **C**.

Our examples already show two things: First, although the semantics of exclusive and parallel splits and joins seems to be informally clear, we have to be very precise about their meaning. In particular, it will be necessary to discuss exclusive joins. Second, the semantics must be defined in such a way that desired properties of the specified control flow can be verified. For instance, for the control flow in Example 3 we want to show that it may lead to a deadlock, i.e. the process may get stuck. In order to fulfil these requirements we will exploit Abstract State Machines (ASMs) for the rigorous definition of semantics

2.3 Abstract State Machines

ASMs are a rigorous, state-based method, where a specification can be considered as an iteration of a parallel execution of a set of rules of the form **IF** ⟨condition⟩ **THEN** ⟨action⟩. Conditions are evaluated on states, which are universal algebras, i.e. sets of functions (resulting from interpretations of function symbols). Actions initiated by the activities, gateways and the start and end events update these states, i.e. change the values of functions at certain locations (arguments). Functions of arity 0 capture the usual concepts of variables and constants in case these functions are declared to be dynamic (updateable) or static (non-updateable), respectively. Thus, functions can be *static*, *dynamic* or *derived*, the dynamic

ones being further classified as *controlled*, *monitored*, *shared*, *in* and *out*. Controlled functions are updated by the process, monitored ones by the environment (e.g. in case of sensors), shared ones by both. In-locations are only read, where out-locations are only updated (as e.g. for incoming and outgoing mail). The theoretical background is the ASM thesis (Yuri Gurevich) according to which each parallel algorithm can be step-by-step simulated by an ASM [2, 3, 13].

In principle, any other equivalent formalism could be used, but ASMs have some advantages. In particular, we will see later that the concept of state is very important to easily capture features, where the semantics cannot be simply defined locally. If e.g. Petri nets [23] as a state-less formalism were used instead, this information would have to be captured in the tokens with the disadvantage of potential redundancy, overly complicated evaluation, and lack of clarity.

The ASM thesis guarantees that business processes can be modelled on any level of abstraction, and the levels can be formally related by means of refinement. As ASMs support unbounded parallelism, this may become helpful, as process instances run in parallel and also activities of each process instance do so. As states of ASMs are abstract, this can be exploited to easily capture more advanced concepts such as counting, priorities, coupling with databases, etc. Furthermore, the formal semantics forms a basis for rigorous methods for quality assurance by means of verification. Last, but not least ASMs have been applied in specifications and correctness proofs (even fully mechanised) for many application areas.

2.4 A Glimpse on ASM Semantics

Let us now sketch the ASM-based definition of semantics for simple control flows. We use a variable `PROCESSES` to capture at all time the (identifiers) of active process instances. Then the semantics of a start event is simply to create a new process instance:

LET $p = \text{New}(\text{process-id})$ IN $\text{PROCESSES} := \text{PROCESSES} \cup \{p\}$

Similarly, the semantics of an end event is to delete each incoming token – we will discuss the token model in the remainder of this section – and to remove a process instance from `PROCESSES`, if no more tokens are left:

IF *no_more_tokens*(p) THEN $\text{PROCESSES} := \text{PROCESSES} - \{p\}$ ENDIF

The basic rules for control flows can now be specified exploiting the parallelism and a token model. As already remarked, process instances run in parallel, which can be defined as follows:

FORALL p WITH $p \in \text{PROCESSES}$ DO $\text{run}(p)$ ENDDO

Furthermore, also flow nodes, i.e. gateways, activities, etc. run in parallel:

FORALL f WITH $f \in \text{FLOW-NODES}(p)$ DO $\text{execute}(f(p))$ ENDDO

For the specification of the control flow execution exploit a token model. For this we first capture the edges (aka sequence flows) in the underlying graph by means of functions *IncomingSequenceFlows* and *OutgoingSequenceFlows*, both

defined on $\text{FLOW-NODES}(p)$. Then we use an edge labelling function *tokensInSequenceFlow* associating a set of tokens with each sequence flow. Tokens can be modelled just by identifiers, but in addition we may define functions associating more detailed information with a token such as the associated process instance, the creating flow node, etc.

The semantics for flow nodes can then be specified in general as follows:

rule NodeTransition(flowNode) =

IF

controlCondition(flowNode) **AND** eventCondition(flowNode) **AND**
dataCondition(flowNode) **AND** resourceCondition(flowNode)

THEN

parblock

controlOperation(flowNode)
eventOperation(flowNode)
dataOperation(flowNode)
resourceOperation(flowNode)

endparblock

For the core model only the controlCondition and the controlOperation are relevant. This gives: $\text{execute}(f(p)) = \text{NodeTransition}(f(p))$.

Specialisation for Split Gateways. Let us now take a closer look at split gateways. For both exclusive and parallel split gateways the control condition is the same:

splitControlCondition(flowNode) =
Exists $e \in \text{IncomingSequenceFlows}(\text{flowNode})$
With $\text{tokensInSequenceFlow}(e) \neq \emptyset$

Also, in both cases one incoming token has to be removed:

removeIncomingToken(flowNode) =
Choose $e \in \text{IncomingSequenceFlows}(\text{flowNode})$
With $\text{tokensInSequenceFlow}(e) \neq \emptyset$
Choose $t \in \text{tokensInSequenceFlow}(e)$
 $\text{tokensInSequenceFlow}(e) := \text{tokensInSequenceFlow}(e) - \{t\}$

However, the production of outgoing tokens differs for exclusive and parallel split gateways. For exclusive split gateways just one outgoing token is produced

produceOneOutgoingToken(flowNode) =
Choose $e \in \text{OutgoingSequenceFlows}(\text{flowNode})$
Let $t = \text{New}(\text{token})$ **In**
 $\text{tokensInSequenceFlow}(e) := \text{tokensInSequenceFlow}(e) \cup \{t\}$

For exclusive split gateways one token is produced for every outgoing sequence flow:

produceAllOutgoingToken(flowNode) =
Forall $e \in \text{OutgoingSequenceFlows}(\text{flowNode})$ **DO**

Let $t = \mathbf{New}(\text{token})$ **In**
 $\text{tokensInSequenceFlow}(e) := \text{tokensInSequenceFlow}(e) \cup \{t\}$
ENDDO

In summary, we obtain the following refinement for the exclusive split gateway:

$\text{exclusiveSplitTransition}(\text{flowNode}) = \text{NodeTransition}(\text{flowNode})$ **Where**
 $\text{controlCondition}(\text{flowNode}) = \text{splitControlCondition}(\text{flowNode})$ **AND**
 $\text{controlOperation}(\text{flowNode}) =$
parblock
 $\text{removeIncomingToken}(\text{flowNode})$
 $\text{produceOneOutgoingToken}(\text{flowNode})$
endparblock

Analogously, we obtain the following refinement for the parallel split gateway

$\text{parallelSplitTransition}(\text{flowNode}) = \text{NodeTransition}(\text{flowNode})$ **Where**
 $\text{controlCondition}(\text{flowNode}) = \text{splitControlCondition}(\text{flowNode})$ **AND**
 $\text{controlOperation}(\text{flowNode}) =$
parblock
 $\text{removeIncomingToken}(\text{flowNode})$
 $\text{produceAllOutgoingToken}(\text{flowNode})$
endparblock

Specialisation for Join Gateways. For exclusive join gateways the control condition is the same as for the split:

$\text{exclusiveJoinControlCondition}(\text{flowNode}) =$
Exists $e \in \text{IncomingSequenceFlows}(\text{flowNode})$
With $\text{tokensInSequenceFlow}(e) \neq \emptyset$

For parallel join gateways the control condition tokens must exist for all incoming sequence flows:

$\text{parallelJoinControlCondition}(\text{flowNode}) =$
All $e \in \text{IncomingSequenceFlows}(\text{flowNode})$
With $\text{tokensInSequenceFlow}(e) \neq \emptyset$

In both cases just one outgoing token is produced, for which the operation $\text{produceOneOutgoingToken}(\text{flowNode})$ can be reused.

For exclusive join gateways just one ingoing token is removed:

$\text{removeOneIncomingToken}(\text{flowNode}) =$
Choose $e \in \text{IncomingSequenceFlows}(\text{flowNode})$
With $\text{tokensInSequenceFlow}(e) \neq \emptyset$
Choose $t \in \text{tokensInSequenceFlow}(e)$
 $\text{tokensInSequenceFlow}(e) := \text{tokensInSequenceFlow}(e) - \{t\}$

For parallel join gateways one token is removed for every incoming sequence flow:

```
removeAllIncomingToken(flowNode) =
  Forall  $e \in \text{IncomingSequenceFlows}(\text{flowNode})$  Do
    Choose  $t \in \text{tokensInSequenceFlow}(e)$ 
       $\text{tokensInSequenceFlow}(e) := \text{tokensInSequenceFlow}(e) - \{t\}$ 
    Enddo
```

In summary, we obtain the following refinement for the exclusive join gateway:

```
exclusiveJoinTransition(flowNode) = NodeTransition(flowNode) Where
controlCondition(flowNode) = exclusiveJoinControlCondition(flowNode) AND
controlOperation(flowNode) =
  parblock
    removeOneIncomingToken(flowNode)
    produceOneOutgoingToken(flowNode)
  endparblock
```

Analogously, we obtain the following refinement for the parallel join gateway:

```
parallelJoinTransition(flowNode) = NodeTransition(flowNode) Where
controlCondition(flowNode) = parallelJoinControlCondition(flowNode) AND
controlOperation(flowNode) =
  parblock
    removeAllIncomingToken(flowNode)
    produceOneOutgoingToken(flowNode)
  endparblock
```

Let us finally look again at the semantics specified above for the exclusive join gateway. Roughly said, the specified semantics is “one incoming token will be removed and one outgoing token will be produced.” Actually, this is equivalent to doing nothing: each token appearing on any incoming sequence flow is simply forwarded to the outgoing sequence flow – control flow is simply passed on. As we saw in Example 2 the effect may be that follow-on activities must be executed multiple times.

So the question is, whether this “empty” semantics is really the desired one? Alternatives to the specified semantics are the following ones:

- **True Join:** Only one incoming token is considered, all others are ignored, i.e. “the first one will be served”. In this case other incoming tokens have to be deleted including those that still may arrive, but not via loops, so it is getting tricky, and additional information has to be kept in the state. We will discuss this alternative in the contexts of inclusive join gateways and flags in Sect. 3.
- **True XOR:** If it is possible to have more than one incoming token, this is considered an error. In this case it has to be verified that the erroneous situation may never occur.

2.5 A Glimpse on Verification

In our discussion of semantics we have now seen already several cases, where it is necessary to verify desired properties of control flow specifications. Examples of such properties are the following:

- **Liveness:** Each started process will eventually terminate.
- **No deadlocks:** A flow flode may have to wait forever, though the process instance is not yet completed.
- **No redundancy:** Each activity or gateway can eventually be fired.
- **True XOR join:** Only one token may arrive at an incoming sequence flow of an exclusive gateway (third semantics).

Example 4. In Examples 1–3 we should see the following properties:

- Liveness holds for the first three cases (empty semantics assumed) in Figs. 1, 2 and 3, only in the first two cases for the “true XOR” semantics, but not in the fourth case in Fig. 4.
- There are no deadlocks in the first three cases and no redundant flow nodes in any of the four cases.
- In the third case two tokens may arrive at the exclusive join, while in the fourth case it may occur that only one token arrives at the parallel join thus creating a deadlock.

In all these cases the proofs are rather obvious. Nonetheless, let us sketch examples of liveness proofs.

First consider the control flow in Example 1 corresponding to Fig. 1. It can be easily seen that at any time there is at most one token, and that each path starting at the start node leads to the end node.

Next consider the slightly more complicated control flow in Example 1 corresponding to Fig. 2. In this case, if the parallel split is fired, then also the parallel join will fire. Then the proof can be reduced to the argumentation for the first example.

3 Control Flow Extensions

The discussion of different semantics for exclusive joins motivates to think about alternative gateways for synchronisation. Therefore, in this section we will introduce a few extensions to the simple control flow specifications considered so far. We will start with a discussion of inclusive split and join gateways. Informally, an *inclusion split* gateway will produce tokens on any subset of outgoing sequence flows, which can be easily specified by ASMs. Then an *inclusive join* must synchronise all incoming tokens that may arrive, but those that may arrive via a loop have to be excluded.

We adopt the semantics defined by Börger, Sörensen and Thalheim in [6], which provides a nice example for the use of flags in the state. Flags are de facto Boolean-valued locations, which can be used to define more complex control conditions and actions. As another extension we briefly sketch counters, i.e. integer-valued locations that can be used among others to capture priorities.

3.1 Inclusive Gateways

As stated above an inclusive join gateway has to synchronise all incoming tokens that may arrive (excluding loops). So, we need a different control condition. However, as the condition has to capture, if any token may still arrive, the property is no longer local, i.e. it does not only depend on the token on incoming edges.

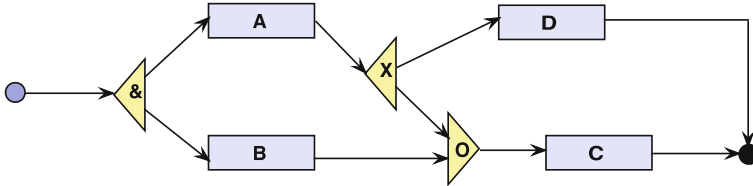


Fig. 5. Control flow with inclusive join gateway

Example 5. Let us modify the (erroneous) Example 3 illustrated in Fig. 4 by replacing the parallel join by an inclusive one. This is illustrated in Fig. 5. Intuitively, the informal semantics sketched above indicates that the control flow is now correct. As before after start the activities **A** and **B** will be executed in parallel and eventually completed. So the inclusive join will receive a token from **B** and wait, if another token arrives from the exclusive split following the completion of **A**. Now, if the exclusive split enables activity **D**, no such token may arrive at the inclusive join anymore, i.e. it will fire and thus enable activity **C** – so in this case both **C** and **D** will be executed. Otherwise, if the decision at the exclusive split is different, then **D** will not be executed, but the second token will appear at the other incoming edge of the inclusive gateway, which will fire and enable activity **C** – so in this case only **C** will be executed.

Example 6. Now consider a more complicated example as illustrated by the control flow in Fig. 6. After start the inclusive split gateway will select any subset of activities **A**, **B** and **C** to be executed in parallel. Regardless, which of these activities have been selected, they will eventually terminate. If **B** or **C** or both were selected, this will be synchronised again by the first inclusive join, and the

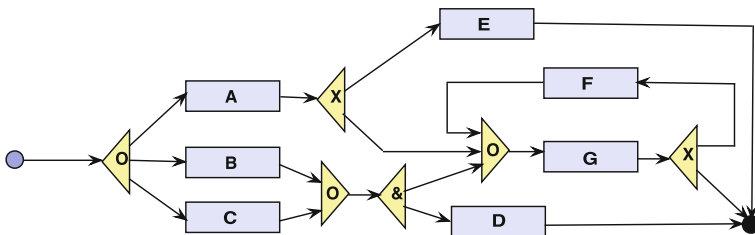


Fig. 6. Control flow with inclusive split and join gateways and feedback loop

follow-on parallel split will then pass tokens onto activity **D** and the second inclusive join. The latter one has two more incoming sequence flows. The one coming from activity **F** has to be ignored, as a token can only come this way, if it had passed already the (waiting) inclusive gateway, which is impossible. So, if **A** has been selected, executed, and the follow-on exclusive split does not select to pass on control to activity **E**, but to the second inclusive gateway, this one will fire.

To specify the semantics of the inclusive gateway it is therefore possible to proceed as follows. The inclusive split at the beginning “informs” all possible successors that are inclusive joins about its decision. That is, if **A** is in the selected set, the second inclusive gateway will receive a flag on the incoming flow corresponding to the path through **A**. Analogously, if one or both of **B** or **C** is selected, the other incoming flow of the second inclusive join will receive a flag in addition to flags on the incoming edges of the first inclusive gateway. If **A** is completed, but the following exclusive split passes control onto **E**, the corresponding flag can be removed, as no token may arrive via this path. Then an inclusive join can fire, if all tokens on incoming edges that could arrive actually have arrived. Then all tokens and flags will be removed and a token at the outgoing edge will be created.

Let us finally remark on the feedback loop. In case the second inclusive gateway fires, first **G** will be enabled. Then after completion, **F** might be enabled, in which case also a flag is created at the third incoming edge of the inclusive join. As no other flag could be set, this gateway could immediately fire again.

3.2 Semantics of Inclusive Gateways Using Flags

For inclusive splits the changes to exclusive or parallel gateways, respectively, are straightforward. Instead of creating exactly one token on one (or all, respectively) outgoing sequence flows, we select an arbitrary (in general non-empty) subset of these sequence flows and create tokens on all of them. In addition, all split gateways will have to create control flags for “reachable” inclusive join gateways, as we will discuss next.

So, for inclusive joins we want to exploit the paths from a split gateway of any type to a join gateway without any repetition of flow nodes. For this we can use a function *reachableJoin* defined for all outgoing sequence flows of split gateways assigning a set of sequence flows to them. More precisely, if there is a path from a flow node f to an inclusive join gateway f_o (not involving f_o itself) with initial sequence flow e (which is an outgoing sequence flow of f), then the final sequence flow e' (which is an incoming sequence flow of f_o) is in *reachableJoin*(e). The function *reachableJoin* is static and defined only by the control flow.

The idea is then that whenever a token is placed on an outgoing sequence flow e , a *flag* is defined for all $e' \in \text{reachableJoin}(e)$ indicating that a token may arrive this way. These flags can be updated, so the information, which tokens may still arrive is kept up to date. Thus, for each incoming sequence flow e of an inclusive join we define a function *flags* resulting in a set of flags, where a

flag is defined as $t.o$, where t identifies a token, and o indicates the flow node, at which the token was generated. Then we have to refine all previous definitions of gateways:

- For each flow node the origin o of the generated tokens $t.o$ is an identifier for the gateway.
- All flags $t.o$ consumed by flow node f appearing in some $flags(e')$ with $e' \in reachableJoin(e)$ and $e \in OutgoingSequenceFlows(f)$ will be replaced by a new flag $t'.f$, where t' is the new token produced on e .
- In particular, at the start s the generated token $t.s$ will be placed into all $flags(e')$ with $e' \in reachableJoin(s)$.

The refinement of the ASM specifications is straightforward.

Let us now look at the definition of the control operation and condition for inclusive join gateways. Informally, the control operation for an inclusive join is analogous to exclusive and parallel joins, as only one token is generated and the tokens needed in the control condition are removed. This includes the deletion of the flags. The control condition is simply to check that all tokens that may arrive have actually arrived:

inclusiveJoinControlCondition(flowNode) =
All $e \in IncomingSequenceFlows(flowNode)$
With $tokensInSequenceFlow(e) = flags(e)$

More details concerning the ASM specification for inclusive gateways can be found in [6]. It should be noted that in this particular case concerning the specification of inclusive joins the use of flags can be avoided as the inclusiveJoinControlCondition can be derived from the distribution of tokens, i.e. from the state. Furthermore, if multisets are used, identifiers for tokens may be preserved. However, it should also be noted that in any case the semantics of the inclusive gateway is no longer “local”, as tokens in different parts of the specification have to be explored. This is simplified by the use of flags.

Nonetheless, flags associated with incoming (and outgoing) sequence flows can be used as a general extension mechanism. Another example for their use could be the definition of semantics of the “true join” semantics for exclusive joins “the first one coming is served”.

3.3 Counters and Priorities

As we have already seen in our discussion of very simple examples (see Fig. 3), already with the (empty) semantics of exclusive joins it is possible to create multiple tokens associated with a single edge. Therefore, it appears natural to associate counters with incoming sequence flows to define complex control conditions requiring specified numbers of tokens on each incoming sequence flow to be consumed. Analogously, we may associate counters with outgoing sequence flows to define complex control operations generating specified numbers of tokens on

each outgoing sequence flow. As such counters belong to the state, they may be subject of updates, e.g. updated by activities.

For instance, define how many reviews will be required by a certain application and how many positive ones will be needed for success. Then integrated split and join gateways as well as gateways with only one incoming and one outgoing sequence flow will be enabled.

This leads to several new verification problems. If complex control conditions involving counters are used, the question arises, if these conditions can always be satisfied. Another question is what happens with remaining tokens (as for exclusive joins). These tokens could be ignored or removed. Then again flags are needed for potentially more arriving tokens. Alternatively, left over tokens might be considered as a specification error. In case the soundness can only be checked at run-time, because counters can be updated, an exception may be raised.

Counters may also be used to count how often a flow node has fired for a particular process instance. This can be used to model different behaviour for the first, second, etc. run through the flow node. A special case for the use of such a run counter is the complex gateway in BPMN, for which the internal “state” can be modelled by the run counter, and control operations may affect the counter as well.

In a complex gateway with several incoming and several outgoing sequence flows priorities among several control conditions can be easily modelled. As the semantics of gateways (and other flow nodes) is defined by means of ASMs, any other complex conditions for splitting and synchronising the control flow can be modelled – if necessary, the state signature has to be extended.

4 Messages and Events

Messages and events are necessary extensions in business processes to fine-tune the specification. Basically, a *message* is defined by a *sender*, one or more *receiver(s)* and the *message content*. Sending a message is nothing more than a specific activity, but receiving a message can also be modelled as a specific activity, and the content of a message is created by the sender before sending – this is illustrated by the specific graphical notation for send and receive actions in Fig. 7. We distinguish a *signal* from a message by the main criterium that the signal is created continuously while sending. Messaging can be done in a *synchronous* or *asynchronous* way. Analogously, signals may be transmitted in a *synchronous* way or *spooled*.

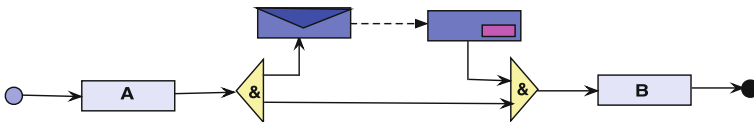


Fig. 7. Send and receive actions in a control flow

As sending and receiving are activities, they appear in control flows just as all other activities. Furthermore, sending and receiving (as activities) produce tokens when executed, and they may change the state in other ways. This has to be distinguished from the events that occur in connection with messaging.

The sending of a message creates an event SENT, and the receiving of a message creates an event RECEIVED. Besides these further message-related events can occur: DELAYED, LOST, REJECTED, UNDELIVERABLE, etc. It seems not advisable to create specific notation for each of these events, in particular not, if messaging events are integrated with other events using a temporal language for creating complex events.

4.1 Message Processing

A detailed ASM specification of communication actions (send, receive) has been done for S-BPM [11]. We adopt this specification in our modelling approach. In this model each receiver is equipped with an *inputPool*, which is subject to size restrictions concerning the total capacity (number of messages), the number of messages from a particular sender, the number of messages of a particular type, and the number of messages of a particular type coming from a particular sender. The bound 0 is used to indicate *synchronous communication*, otherwise it is *asynchronous*. Different strategies for handling violations to these bounds have been specified.

For sending and receiving the actor (sender or receiver) chooses between several alternatives defined by receiver and message type, and prepares a *messageToBeHandled*. For sending this is a *messageToBeSent*, for receiving this is an *expectedIncomingMessage*. Then *TryAlternative_{commAct}* actually tries to send or receive the message.

Let us now sketch the ASM specification for message specification. Details can be found in the appendix of [11].

```

Perform (actor, CommAct, state) =
IF NonBlockingTryRound(actor, state) THEN
    IF TryRoundFinished(actor, state) THEN
        InitializeBlockingTryRounds (actor, state)
    ELSE TryAlternativeCommAct (actor, state) ENDIF
ENDIF
IF BlockingTryRound(actor, state) THEN
    IF TryRoundFinished (actor, state) THEN
        InitializeRoundAlternatives (actor, state)
    ELSIF Timeout (actor, state, timeout (state)) THEN
        InterruptCommAct (actor, state)
    ELSIF UserAbrupton(actor, state) THEN
        AbruptCommAct (actor, state)
    ELSE TryAlternativeCommAct (actor, state) ENDIF
ENDIF
with

```


$TryAlternative_{CommAct}(\text{actor}, \text{state}) =$
 $ChoosePrepareAlternative_{CommAct}(\text{actor}, \text{state}) \text{ seq } Try_{CommAct}(\text{actor}, \text{state})$

In a simple messaging model receivers are well-defined activities, i.e. the actors associated with the receiving activities. However, such a view is rather static. More generally, the state of the processes can be exploited to define receivers in a much more dynamic way. In particular, the receivers may depend on control, event, data and resource conditions just like flow nodes, and the dynamic determination of receivers can be part of the data operations.

4.2 Types of Events

There is a vast amount of possible event types: internal events, timing events, messaging events, etc. In general, events refer to something atomic that “has happened”.

Internal events can refer to particular progress in the process flow, e.g. activities may be enabled, started, completed, but also postponed, interrupted, cancelled, delegated, delayed, etc., gateways may be waiting, enabled, fired, etc., and the whole process may have started or ended.

Likewise messages may have been sent, received, rejected, lost, retried (*n*'th time), etc., and signals may have been started, receiving, received, spooled, etc.

External events refer to activities that are happening in the environment, which can be captured by *monitored functions* in the ASM specification. Events can be combined to *complex events* using the usual logical junctors.

Events can also refer to *time*, for which a discrete, linear time model can be used. In distributed systems it is advisable to refer to local time (time depends on location) in order to avoid the problem of clock synchronisation. With time complex events can be created using temporal relationships: (directly) before, (directly) after, simultaneous, etc. Logically, time events and internal events can be combined.

4.3 Event Conditions and Actions

Thus, the main purpose of events is to enable the specification of *event conditions*, with which the firing of flow node instances becomes dependent on the progress of the process instance. Event conditions have been foreseen for all types of flow nodes, even starting or termination of a process (instance) can be made dependent on an event.

However, event conditions (as specified so far) refer to the firing of flow nodes. Nonetheless, event conditions can also be used to refine the control operation, e.g. deciding, on which outgoing sequence flows tokens shall be created. An example of the latter behaviour is captured by event-based gateways in BPMN. However, using events in the state permits more general definitions of complex gateways or event-driven activities.

Event actions have also been foreseen for all types of flow nodes. The purpose of event actions is to record those events that are relevant for the continuation of

the process, as not all events are relevant to be recorded. That is, the recording would cover which event happened where and when, etc.

Some events may be disruptive requiring the process to be interrupted, (partially) rolled back and restarted with additional information. The handling of interrupts refers to exception handling.

5 Interaction and Data Handling

Interaction refers to the detailed specification, how basic activities are executed. The key idea is that each basic activity defines a *dialogue*, which may be broken down into several *dialogue steps*. We will first look into dialogue specifications, then briefly address how data conditions and actions can be specified. For the latter it is decisive that in each dialogue step certain data are consumed, while other data is produced. Here *data consumption* refers to the data that is needed (i.e. read) to perform any operation associated with the dialogue step, while *data production* refers to the data that is created by one of the operations associated with the dialogue step.

5.1 Interaction and Dialogues

Data consumption and production provide a local view on the data. Thus, both can be defined by small schemata associated with activities and then also dialogue steps. The integration of all these schemata defines a part of a global schema that underlies the data handling of the whole process for all instances.

Therefore, we distinguish between *database objects* defined by the global schema and *dialogue objects* defined by local schemata. Naturally, the local schemata define *views* on the global schema. By abstraction from individual activities we obtain a model of dialogue types, which we adopt from [28].

According to the previous discussion let us assume a (*global*) *database schema*. Elements appearing in instances of the global schema are referred to as *database objects*. These may be relations, trees, graphs, arrays, etc. We deliberately leave this open in order to stress that any data model (relational, nested, object-oriented, tree-based, graph-based, etc.) may be used here, but we must assume a *query language* that can be used to define *views*.

Then a *dialogue type* is basically defined by such a view plus operations. A *dialogue object* is an element in such a view for a particular database instance plus the operations of the dialogue type restricted to this element.

Based on the database schema operations (usually transactions) can be defined. Such *db-operations* require some input types and a specification of the actual updates of the database by means of db-programs.

Analogously, operations on dialogue types can be defined. Such *dialogue-operations* also require input types plus a *selection type* defined on the view and a *body* that specifies which db-operations are used and which other dialogue objects are to be created.

Usually, on a high level of abstraction the concrete definition of the database schema, the queries defining the views and the db-operations are left abstract. According to our discussion a basic activity can be defined by a set of dialogue types defining implicitly data consumption and production and the flow of dialogue steps.

5.2 Data Conditions and Actions

The assumed global schema can be used to define data conditions for all flow nodes in the control flow specification. Such *data conditions* are expressed as Boolean queries. Analogously, the db-operations define *data actions* that may also be associated with the flow nodes in the control flow.

However, for activities the data actions result from the defining dialogue operations. Only the initialisation of the starting dialogue type has to be defined.

Data actions have global effects on the state. In particular, data conditions in remote parts of the specified process now depend on activities that seem to be independent. Thus, data conditions and actions require additional consistency verification due to these hidden dependencies. Furthermore, dialogue operations may also be defined directly on the views requiring a translation of the view updates to database updates. How to verify consistency with respect to data dependencies, is still a matter of research and will not be stressed further in this article.

6 Actors, Roles and Exceptions

The flow-centric specification of business processes emphasises the activities, their effects, and controlling conditions. Those who have to perform the (basic) activities are referred to as *actors*. Thus, to complete the picture, actors have to be associated with all activities. Then it is the *responsibility* of the actor to perform the activity using the associated dialogue objects as tools.

6.1 Responsibilities and Decision Making

Therefore, instead of directly associating actors, activities are usually assigned to *roles*, which are names representing an assortment of *obligations* and *rights*. In particular, some decisions on the flow of a process depend on data and events, while others have to be made explicitly by actors in certain roles.

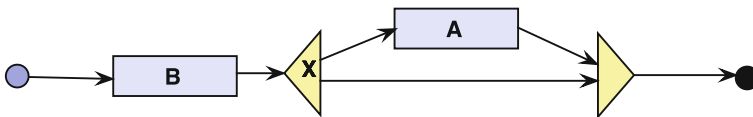


Fig. 8. Control flow with optional activity

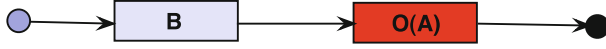


Fig. 9. Control flow with explicitly marked optional activity

Example 7. In the control flow illustrated in Fig. 8 the activity **A** is optional. The question is who decides, if **A** is executed or not?

- Activity **B** may contain a data action such that the follow-on exclusive split can evaluate a data condition, which determines, if **A** or nothing is done. In this case, actually the actor associated with **B** is responsible for the decision.
- Alternatively, the data condition associated with the gateway can be defined elsewhere, if embedded in a larger control flow.
- We may associate a role also with the gateway, which, however, would blur the distinction between gateways that fire immediately when enabled and activities that usually depend on actor interaction.
- Finally, the decision may be a right of the actor associated with activity **A**, in which case the control flow is misleading.

In order to cope with situations as in Example 7 we have to distinguish cases, where decisions are based on pure data conditions from those, where it is the responsibility of the actor him/herself to decide about executing an optional activity. Therefore, deontic rules should be supported, some of which can be easily marked in control flows (see [20,22] for details).

Example 8. For instance, in Example 7 the activity **A** could be marked as optional as illustrated in Fig. 9. Executing **O(A)** means that the actor associated with this activity either executes **A** or nothing. Analogously, alternatives and forbidden actions can be handled this way [20,22].

A more fine-grained specification of rights and obligations associated with roles can be obtained by *deontic action logic*. The *atoms* of the logic are defined as follows:

- $\text{do}(r, a)$ means that an actor in role r *executes* action a .
- $\text{Pdo}(r, a)$ means that an actor in role r is *permitted* to execute action a .
- $\text{Fdo}(r, a)$ means it is *forbidden* for an actor in role r to execute action a .
- $\text{Odo}(r, a)$ means that an actor in role r is *obliged* to execute action a .

Formulae in the logic are constructed from the atoms with the usual logical junctors. Then deontic constraints give rise to another verification task, i.e. to check if processes can be executed under the given constraints.

6.2 Exception Handling

An *exception* is a disruptive event, i.e. an exception causes that the running process instance is interrupted regardless of its state, a complete or partial roll-back to a consistent state will be executed, and depending on the kind of the

exception a restart from a consistent state with a different continuation process will be launched. Examples of exceptions are order cancellation, bancruptcy of a customer, serious breakdown, etc.

The rollback may affect a single or a few activities, but also the complete process may have to be rolled back. The rollback triggered by an exception is a process in its own right. It will first collect activities that have to be undone, either perform **undo** operations as in databases or apply compensation activities if possible. **Undo** operations may require adequate data, i.e. the detailed specifications of activities, gateways, decisions, etc. have to be refined.

At the end of the rollback the process should have reached a consistent previous state, from which the process instance will resume, i.e. continue in the modified state, which also contains information about the exception.

Continuation processes can be part of the process specification. Thus, we may use conservative extensions to specify alternative paths in case of particular exceptions:

```
IF notInterrupted THEN ⟨as specified⟩
      ELSIF exception1 THEN continuation1 ...
```

Usually, continuation processes give rise to subprocesses.

Exception handling is still subject to research, so we dispense with discussing further details in this article.

7 Conclusions

In this article we briefly outlined business process specifications grounded in horizontal refinements, which enables the integration of several sub-systems addressing control flow, message handling, event handling, data handling, exception handling, etc. While some of the work has reached already a mature state, other parts are still under investigation. The final goal is to complete the specification of an integrated business process model H-BPM with unambiguous semantics defined by ASMs.

The main emphasis of H-BPM is to capture all relevant aspects of BPM and to formally define the semantics using Abstract State Machines in order to enable validation and formal verification. Usually actors and data handling have been neglected, while other constructs handled superficially (see [16] for a discussion of major parts of BPMN 2.0). In this way H-BPM aims to enable seamless modelling of business processes on all levels of abstraction as well as horizontal and vertical model integration by means of formal refinement.

Regarding vertical integration is achieved by further refining the involved ASMs in a development process that is targeting the executable specification of a workflow engine that is enriched with components for data and dialogue handling and exception processing. Throughout the process rigorous quality assurance methods have to be applied.

In this brief article we only gave an overview of the most relevant aspects of H-BPM without going too much into details. In particular, several extensions were not yet handled such as modularity by subprocesses with replacement

semantics, which will have many implications on the constructs introduced so far, further details of the ASM specifications, multiple view presentations, in particular coarse and fine presentation of specifications, etc. We also did not stress how to handle validation nor how to effectively perform verification.

Regarding our future research further extensions are envisioned. These comprise the possibility to postpone or cancel activities with the corresponding meta-level rights and obligations. Furthermore, we intend to investigate adaptivity in business process specifications, in particular with respect to preferences, exception handling and ad-hoc changes. Adaptivity is the subject of a running research project AdaBPM.

References

1. Abramowicz, W., Filipowska, A., Kaczmarek, M., Kaczmarek, T.: Semantically enhanced business process modelling notation. In: Hepp, M., et al. (eds.) S-BPM. CEUR Workshop Proceedings, vol. 251 (2007). CEUR-WS.org
2. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. *ACM Trans. Comput. Log.* **4**(4), 578–651 (2003)
3. Blass, A., Gurevich, Y.: Abstract State Machines capture parallel algorithms: Correction and extension. *ACM Trans. Comput. Log.* **9**(3) (2008)
4. Börger, E.: Approaches to modeling business processes: a critical analysis of BPMN, workflow patterns and YAWL. *Softw. Syst. Model.* **11**(3), 305–318 (2012)
5. Börger, E., Sörensen, O.: BPMN core modeling concepts: inheritance-based execution semantics. In: Embley, D., Thalheim, B. (eds.) *Handbook of Conceptual Modeling: Theory, Practice and Research Challenges*, pp. 287–335. Springer, Heidelberg (2011)
6. Börger, E., Sörensen, O., Thalheim, B.: On defining the behavior of OR-joins in business process models. *J. Univ. Comput. Sci.* **15**(1), 3–32 (2009)
7. Börger, E., Stärk, R.: *Abstract State Machines*. Springer, Heidelberg (2003)
8. Börger, E., Thalheim, B.: A Method for verifiable and validatable business process modeling. In: Börger, E., Cisternino, A. (eds.) *Advances in Software Engineering*. LNCS, vol. 5316, pp. 59–115. Springer, Heidelberg (2008)
9. Börger, E., Thalheim, B.: Modeling workflows, interaction patterns, web services and business processes: The ASM-based approach. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) *ABZ 2008*. LNCS, vol. 5238, pp. 24–38. Springer, Heidelberg (2008)
10. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: *Fundamentals of Business Process Management*. Springer, Heidelberg (2013)
11. Fleischmann, A., et al.: *Subject-Oriented Business Process Management*. Springer, Heidelberg (2012)
12. Geist, V.: *Integrated Executable Business Process and Dialogue Specification*. Ph.D. thesis, Johannes Kepler University Linz, Austria (2011)
13. Gurevich, Y.: Sequential Abstract State Machines capture sequential algorithms. *ACM Trans. Computat. Log.* **1**(1), 77–111 (2000)
14. Kopetzky, T., Geist, V.: Workflow charts and their precise semantics using abstract state machines. In: Rinderle-Ma, S., Weske, M. (eds.) *Proceedings of EMISA 2012 - Der Mensch im Zentrum der Modellierung*, Vienna, Austria (2012). LNI, pp. 11–24. Kllen-Verlag, Bonn (2012)

15. Kossak, F., Illibauer, C., Geist, V.: Event-based gateways: open questions and inconsistencies. In: Mendling, J., Weidlich, M. (eds.) BPMN 2012. LNBIP, vol. 125, pp. 53–67. Springer, Heidelberg (2012)
16. Kossak, F., et al.: A Rigorous Semantics for BPMN 2.0 Process Diagrams. Springer (2014, forthcoming)
17. Natschläger, C.: Deontic BPMN. In: Hameurlain, A., Liddle, S.W., Schewe, K.-D., Zhou, X. (eds.) DEXA 2011, Part II. LNCS, vol. 6861, pp. 264–278. Springer, Heidelberg (2011)
18. Natschläger, C.: Towards a BPMN 2.0 ontology. In: Dijkman, R., Hofstetter, J., Koehler, J. (eds.) BPMN 2011. LNBIP, vol. 95, pp. 1–15. Springer, Heidelberg (2011)
19. Natschläger, C., Geist, V., Kossak, F., Freudenthaler, B.: Optional activities in process flows. In: Rinderle-Ma, S., Weske, M. (eds.) Der Mensch im Zentrum der Modellierung. LNI. Kllen-Verlag, Bonn (2012)
20. Natschläger, C., Kossak, F., Schewe, K.D.: BPMN to Deontic BPMN: A trusted model transformation. *Journal of Software and Systems Modelling* (2014, to appear)
21. Natschläger, C., Schewe, K.D.: A flattening approach for attributed type graphs with inheritance in algebraic graph transformation. *Electron. Commun. EASST* **47**, 160–173 (2012)
22. Natschläger-Carpella, C.: Extending BPMN with Deontic Logic. Logos Verlag, Berlin (2012)
23. Petri, C.A.: Communication with automata. Ph.D. thesis, Universität Hamburg (1966)
24. Recker, J.C., Rosemann, M., Indulska, M., Green, P.: Business process modeling: A comparative analysis (2009)
25. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies. Springer, Heidelberg (2012)
26. Scheer, A.W.: ARIS - Business Process Modeling. Springer, Heidelberg (2000)
27. Schewe, K.-D.: Horizontal and vertical business process model integration. In: Decker, H., Lhotská, L., Link, S., Basl, J., Tjoa, A.M. (eds.) DEXA 2013, Part I. LNCS, vol. 8055, pp. 1–3. Springer, Heidelberg (2013)
28. Schewe, K.D., Schewe, B.: Integrating database and dialogue design. *Knowl. Inf. Syst.* **2**(1), 1–32 (2000)
29. ter Hofstede, A.M., et al. (eds.): Modern Business Process Automation: YAWL and its Support Environment. Springer, Heidelberg (2010)
30. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer, Heidelberg (2012)
31. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M., Russell, N.: On the suitability of BPMN for business process modelling. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 161–176. Springer, Heidelberg (2006)
32. Wong, P.Y.H., Gibbons, J.: A process semantics for BPMN. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 355–374. Springer, Heidelberg (2008)
33. zur Muehlen, M., Recker, J.C., Indulska, M.: Sometimes less is more: are process modeling languages overly complex? In: Taveter, K., Gasevic, D. (eds.) 3rd International Workshop on Vocabularies, Ontologies and Rules for the Enterprise. IEEE, Annapolis (2007)

Transactions on Large-Scale Data- and
Knowledge-Centered Systems XVIII
Special Issue on Database- and Expert-Systems
Applications
Hameurlain, A.; Küng, J.; Wagner, R.; Decker, H.;
Lhotska, L.; Link, S. (Eds.)
2015, XI, 207 p. 84 illus., Softcover
ISBN: 978-3-662-46484-7