Journal of Universal Computer Science, vol. 14, no. 1 (2009), 3-32
submitted: 2/9/08, accepted: 27/10/08, appeared: 1/1/09 © J.UCS

On Defining the Behavior of OR-joins in Business Process Models

Egon Börger[1]
(Università di Pisa, Italy
boerger@di.unipi.it)

Ove Sörensen
(University of Kiel, Germany
ove@is.informatik.uni-kiel.de)

Bernhard Thalheim
(University of Kiel, Germany
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 major 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 allows 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.[2]

Key Words: business processes, OR-join problem, BPMN standard
Category: D.1.7, D.2.1, D.2.4

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

[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 third author at the Computer Science Department of the University of Kiel, Germany.

[2] The simulator has been developed by the second author and is part of his Diplom Thesis [Sörensen, 2008].

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 *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) [A.-W.Scheer, 1994] (see for example the analysis in [van der Aalst et al., 2002], [Mendling et al., 2006]) and the current BPMN standard [BPMI.org, 2006] (see for example the analysis in [Dijkman et al., 2007], [Grosskopf, 2007]). 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 [Gruhn and Laue, 2007] for the fixpoint-semantics-based definition of the semantics of EPCs in [Kindler, 2004], [Kindler, 2005].

The OR-join problem appears in the literature also as a *verification method problem* in the sense that even where a behavioral definition is given, the computational cost of mechanically verifying 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 [van der Aalst and ter Hofstede, 2005], which for reasons of complexity does not consider nested OR-joins. In [Wynn et al., 2006a], which seems to be a reelaboration of [Wynn et al., 2005], two "problems with OR-join semantics as defined in [van der Aalst and ter Hofstede, 2005]" (quote of the title of [Wynn et al., 2006a, Sect.2.2]) are identified and the restriction is eliminated, based upon the work in [Wynn, 2006] (see also [Wynn et al., 2006b]). However, the general solution in [Wynn, 2006] comes with a computational complexity that is considered as too high by the authors of [Dumas et al., 2007] and [Grosskopf, 2007, 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 [Gruhn and Laue, 2005], [Gruhn and Laue, 2006], [Gruhn and Laue, 2007] 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 [Russel et al., 2006].

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 [Börger, 2007a]).

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 happend 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 [van der Aalst,], 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.[3]

Our definition is based upon a framework developed recently in [Börger and Thalheim, 2008], motivated by the goal to define a complete rigorous semantical model for the current BPMN standard [BPMI.org, 2006] and its forthcoming extension 2.0.[4] We start here from scratch and recapitulate a few of the definitions from [Börger and Thalheim, 2008] 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 [Section 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 [Section 3] we sketch the framework that is used in [Section 4] to define a precise semantics for the general intuitive understanding of the OR-join. In [Sörensen, 2008] this definition is extended to provide better structuring for the case when multiple tokens may occur in a cyclic diagram.

To introduce the OR-join model, we will use the technique of stepwise ASM refinement [Börger, 2003]. 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 the next refinement step we consider cyclic workflows. They are BPMN standard conform, but their semantics is under-specified. This 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

[3] Just before submitting the final version of this paper to the editor we found a paper [van Hee et al., 2006] which also proposes to use run-time information for defining the precise behavior of OR-joins. The authors propose 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 [van Hee et al., 2006] only rather special cyclic workflow diagrams can be proved to be without deadlocks.

[4] 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.

flow by associating a *token set* to each cyclic token. This approach is extended in [Sörensen, 2008] 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 [Sörensen, 2008].

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

2 Analysis of OR-Join Requirements

In this section we try to review the intuition 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. [BPMI.org, 2006, 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 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 [Rittgen, 1999] 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 [Section 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 complete[5], 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 [Russel et al., 2006]), 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 [van der Aalst et al., 2003], 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 [Börger, 2007b].

The common feature of the above two and other descriptions in the literature seems to be that some synchronization is to be performed[6] and that this should happen only for currently active threads. The debated question is how to determine whether a thread is (potentially?) active. In [Wynn et al., 2006a, 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 posible to mark more input conditions.

[5] This is a form of the so-called Discriminator pattern in [van der Aalst et al., 2003].

[6] 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.

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.

We advocate to separate the two different concerns involved. We *first* provide in [Section 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 [van der Aalst and ter Hofstede, 2005], [Wynn et al., 2006d], [Russel et al., 2007b] (how often a new one [Russel et al., 2007a]?) or extension of Petri nets [Wynn et al., 2006a], [Wynn et al., 2006b], [Wynn et al., 2006c]. Our definition reflects the global features of the intended synchronization in a direct way, avoiding the well-known problems, discussed for example in [Grosskopf, 2007], 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 [Wynn et al., 2006d], [Wynn et al., 2006b], [Wynn et al., 2006c]. 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

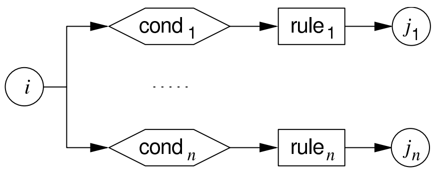
In this section we borrow from [Börger and Thalheim, 2008] that part of the business process modeling framework that allows one to capture the intuition of the OR-join by a concise and clear definition. For the sake of definiteness we use for the discussion of workflow constructs a BPMN-based terminology, 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:

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

where *Updates* stands for a set of *function updates* $f(t_1, \ldots, f_n) := t$ built from expressions t_i, t and an n-ary function symbol f. 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.



if $ctl\_state = i$ **then**
  if $cond_1$ **then**
    $rule_1$
    $ctl\_state := j_1$
    $\ldots$
  if $cond_n$ **then**
    $rule_n$
    $ctl\_state := j_n$

Figure 1: *Viewing (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 [Börger and Stärk, 2003]. 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 [Börger, 2003].

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.

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

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) [Börger and Thalheim, 2008]. For the discussion of the OR-join problem we can focus the discussion on gateways only (see [Section 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 [BPMI.org, 2006, 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 $token(arc)$.[7] 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*.

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

In the token based approach to control, for a rule at a target node of incoming arcs to become fireable some (maybe all) arcs must be enabled. This condition is typically required to be an atomic quantity formula stating that the number of tokens currently associated to *in* (read: the cardinality of $token(in)$, denoted $\mid token(in) \mid$) is at least the input quantity $inQty(in)$ required at this arc.

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

Correspondingly the control operation CtlOp of a workflow usually consists of two parts, one describing how many tokens are Consumed on which incoming arcs and one describing which tokens are Produced on which outgoing arcs in a quantity as indicated by a function $outQty(out)$. 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 Passed from an incoming to an outgoing arc. $outQty(out)$ denotes the number of tokens one wants to be produced on arc *out*. In many applications $inQty(in), outQty(out)$ are assumed to take the default value 1.

$$\begin{aligned}
&\textsc{Consume}(t, in) = \textsc{Delete}(t, inQty(in), token(in))\\
&\textsc{Produce}(t, out) = \textsc{Insert}(t, outQty(out), token(out))\\
&\textsc{Pass}(t, in, out) =\\
&\quad \textsc{Delete}(t, inQty(in), token(in))\\
&\quad \textsc{Insert}(t, outQty(out), token(out))
\end{aligned}$$

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

$$\begin{aligned}
&\textsc{ConsumeAll}(X) = \textbf{forall}\ x \in X\ \textsc{Consume}(x)\\
&\textsc{ProduceAll}(Y) = \textbf{forall}\ y \in Y\ \textsc{Produce}(y)
\end{aligned}$$

Remark This use of macros allows one to easily adapt the abstract token model to its extensions, like the ones we use in [Section 4], and to different instantiations by a concrete token model. For example, if a token is simply

[7] 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.

defined as a pair $(proc(t), pos(t))$ of the *proc*ess instance it belongs to and the arc where it is *pos*itioned, then it suffices to refine the macro for PASSing a token t from in to out by updating the second token component, namely from its current *pos*ition value in to its new value out:

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

The use of abstract DELETE and INSERT operations instead of directly updating $token(a)$ 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 DELETE(in, t) and INSERT(out, t) one may have $in = out$.

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' [BPMI.org, 2006, p.68]. Both splitting and merging come usually in two forms, which are related to the propositional operators **and** and **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**. [BPMI.org, 2006, 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" [BPMI.org, 2006, 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:

$$andSplitToken(t, o) = t.o$$

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 $inArc(node)$.

$$
\begin{array}{l}
\textsc{AndSplitGateTransition}(node) = \textsc{WorkflowTransition}(node)\\
\textbf{where}\\
\quad \textbf{let}\ \{in\} = inArc(node)\\
\quad\quad CtlCond(node) = Enabled(in)\\
\quad\quad \textsc{CtlOp}(node) =\\
\quad\quad\quad \textbf{let}\ t = firingToken(\{in\})\\
\quad\quad\quad\quad \textsc{Consume}(t, in)\\
\quad\quad\quad\quad \textsc{ProduceAll}(\{(andSplitToken(t, o), o) \mid o \in outArc(node)\})
\end{array}
$$

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*.

$$
\begin{array}{l}
\textsc{AndJoinGateTransition}(node) = \textsc{WorkflowTransition}(node)\\
\quad \textbf{where}\\
\quad\quad CtlCond(node) = \textbf{forall } in \in inArc(node)\ Enabled(in)\\
\quad\quad \textsc{CtlOp}(node) =\\
\quad\quad\quad \textbf{let } \{in_1, \ldots, in_n\} = inArc(node)\\
\quad\quad\quad \textbf{let } \{t_1, \ldots, t_n\} = firingToken(inArc(node))\\
\quad\quad\quad\quad \textsc{ConsumeAll}(\{(t_i, in_i)) \mid 1 \leq i \leq n\})\\
\quad\quad\quad\quad \textsc{Produce}(andJoinToken(\{t_1, \ldots, t_n\}), out)
\end{array}
$$

The reader will have noticed that we did not specify the firing tokens by **let** $t_i = firingToken(in_i)$, 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 [Section 3] to define a precise semantics for the general supposedly intuitive understanding of the OR-join. We first define in [Section 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 [Section 4.2] and one for graphs with cycles [Section 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 $select_{Produce}(node)$, which is constrained to select at each invocation a non-empty subset of arcs outgoing *node* that satisfy the *OrSplitCond*ition. 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 $select_{Produce}$
$select_{Produce}(node) \neq \emptyset$
$select_{Produce}(node) \subseteq \{out \in outArc(node) \mid OrSplitCond(out)\}$[8]

This leads to the following instantiation of the $\text{WorkflowTransition}(node)$ scheme for OR-split gateway *node*s. The involvement of process data or gate events for the decision upon the alternatives is formalized by letting *DataCond* and *EventCond* in the rule guard and their related operations in the rule body depend on the parameter O for the chosen set of alternatives. As 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.

$$
\begin{array}{l}
\textsc{OrSplitGateTransition}(node) = \textsc{WorkflowTransition}(node)\\
\quad \textbf{where}\\
\quad\quad \textbf{let } \{in\} = inArc(node)\\
\quad\quad \textbf{let } O = select_{Produce}(node) \textbf{ in}\\
\quad\quad\quad CtlCond(node) = Enabled(in)\\
\quad\quad\quad \textsc{CtlOp}(node, O) =\\
\quad\quad\quad\quad \textbf{let } t = firingToken(\{in\})\\
\quad\quad\quad\quad\quad \textsc{Consume}(t, in)\\
\quad\quad\quad\quad\quad \textsc{ProduceAll}(\{(orSplitToken(t, o), o) \mid o \in O\})
\end{array}
$$

As AndSplitGateTransition is an instance of OrSplitGateTransition, namely with the selection function required to yield the entire set $outArc(node)$, 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 **Unique Start Node Assumption**

[8] 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.

that each graph has exactly one start node.[9] 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 *impossible* to define a formal semantics ... that is fully compliant with the informal semantics" [Gruhn and Laue, 2007, p.6], but as the authors of [Dumas et al., 2007] point out, the problem is not in the *definition* of what they call the OR-join *firing rule*, but in a) the definition of when this rule should be considered as *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 $select_{Consume}$ the mechanism used above to describe via $select_{Produce}$ 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 $select_{Consume}$ 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 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 $select_{Consume}$ 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 $select_{Consume}$. Thus $select_{Consume}(node)$ plays the role of an interface for triggering for a set of to-be-synchronized incoming arcs the execution of the rule at the given *node*, with the usual effect.

$$
\begin{array}{l}
\textsc{OrJoinGateTransition}(node) = \textsc{WorkflowTransition}(node) \\
\quad \mathbf{where} \\
\quad\quad \mathbf{let}\ I = select_{Consume}(node)\ \mathbf{in} \\
\quad\quad\quad CtlCond(node, I) = (I \neq \emptyset\ \mathbf{and\ forall}\ i \in I\ Enabled(i)) \\
\quad\quad\quad \textsc{CtlOp}(node, I) = \\
\quad\quad\quad\quad \textsc{Produce}(orJoinToken(firingToken(I)), out) \\
\quad\quad\quad\quad \textsc{ConsumeAll}(\{(t_i, in_i) \mid 1 \leq i \leq n\})\ \mathbf{where} \\
\quad\quad\quad\quad\quad \{t_1, \ldots, t_n\} = firingToken(I) \\
\quad\quad\quad\quad\quad \{in_1, \ldots, in_n\} = I
\end{array}
$$

The $select_{Consume}$ function in the OrJoinGateTransition serves to express on which arcs one has to wait for tokens of the indicated type from the

[9] 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.

to-be-synchronized threads [van der Aalst et al., 2003], in terms of the BPMN standard document on which arcs we "are expecting a signal based on the upstream structure of the Process" [BPMI.org, 2006, p.81]. The real question is first of all which synchronization condition one wants to impose as constraint on the $select_{Consume}$ function,[10] 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" [Gruhn and Laue, 2007]. In the following we try to investigate some variations of the 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.

4.2.1 "Informal semantics" of OR-join

We start with an analysis of the proposal quoted in [Section 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 [Dumas et al., 2007] which improves on [Wynn et al., 2005]. 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

[10] 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 $select_{Consume}$, instead of (for example) *synchronize*, to show that different interpretations of this function correspond to different choices made for the synchronization discipline at OR-joins.

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 CtlOp,
- add the intended synchronization counterpart $\mathit{CtlCondSync}(node, I)$ to the $\mathit{CtlCond}(node, I)$ in the join node rules, checking whether for each synchronization token an enabling token is present.

Here are the details of this refinement step.

- **Split gate transition refinement.** Let *node* be a split node and *out* any arc outgoing *node* where a token t enabling the unique incoming arc *in* Produces a token $t.out$. This starts a new computation path at *out* that may need to be synchronized with other computation paths started simultaneously at this *node* (or with some final segment of some computation paths started upstream, i.e. at nodes from where *node* can be reached[11]). We place an additional synchronizer copy of $t.out$ on each reachable arc that enters a join node, more precisely for each path that starts with *out* and leads to an arc a entering a join node, we place a synchronizer copy of $t.out$ on a. We denote the set of these join *arc*s by $\mathit{AllJoinArc}(out)$ and record the synchronizer token copy placed there in a location $\mathit{syncToken}(arc)$. This allows us 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 $o \in \mathit{outArc}(node)$ from each $i \in \mathit{AllJoinArc}(o)$. 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 [Section 4.3].
- **Join gate transition refinement.** Let *node* be a join node. The rule 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 $\mathit{CtlCond}(node, I)$ by adding the intended synchronization condition *CtlCondSync*. In the case of the informal OR-join semantics we are formaliz-

[11] 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 [BPMI.org, 2006]. It is avoided for example in Occam-like OR-join interpretations discussed below.

ing 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*.[12] 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.

$$\begin{aligned}
&\textsc{ProduceSync}(t, \mathit{in}) = \ \textsc{Insert}(t, \mathit{syncToken}(\mathit{in}))\\
&\textsc{ConsumeSync}(t, \mathit{in}) = \ \textsc{Delete}(t, \mathit{syncToken}(\mathit{in}))\\
&\textsc{ProduceSyncAll}(Y) = \textbf{forall } y \in Y \ \textsc{ProduceSync}(y)\\
&\textsc{ConsumeSyncAll}(X) = \textbf{forall } x \in X \ \textsc{ConsumeSync}(x)
\end{aligned}$$

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*.

$$\begin{aligned}
&\textsc{ProduceSyncAll}(\{(t.o, i) \mid i \in \mathit{AllJoinArc}(o), o \in O\})\\
&\textsc{ConsumeSyncAll}(\{(t, i) \mid i \in \mathit{AllJoinArc}(o) \textbf{ forsome } o \in \mathit{outArc}(\mathit{node})\})
\end{aligned}$$

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.

$$\begin{aligned}
&\textsc{ProduceSyncAll}(\{(\mathit{joinToken}(t_1, \ldots, t_n), \mathit{in}) \mid \mathit{in} \in \mathit{AllJoinArc}(\mathit{out})\})\\
&\textsc{ConsumeSyncAll}(\{(t_i, \mathit{in}) \mid \mathit{in} \in \mathit{AllJoinArc}(\mathit{out}), 1 \leq i \leq n\} \cup \{(t_i, \mathit{in}_i) \mid\\
&1 \leq i \leq n\})
\end{aligned}$$

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:

$$\textbf{forall } 1 \leq i \leq n \ t_i = t.\mathit{rest}_i \in \mathit{token}(\mathit{in}_i) \cap \mathit{syncToken}(\mathit{in}_i).$$

[12] The reader who knows the ASM refinement method [Börger, 2003] 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.

Correspondingly we refine *orJoinToken* and *andJoinToken*, respectively, 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.

$CtlCondSync(node, I) =$
  forall $i \in I$ $syncToken(i) \neq \emptyset$ **and**
    forall $i \in inArc(node) \setminus I$ $syncToken(i) = \emptyset$

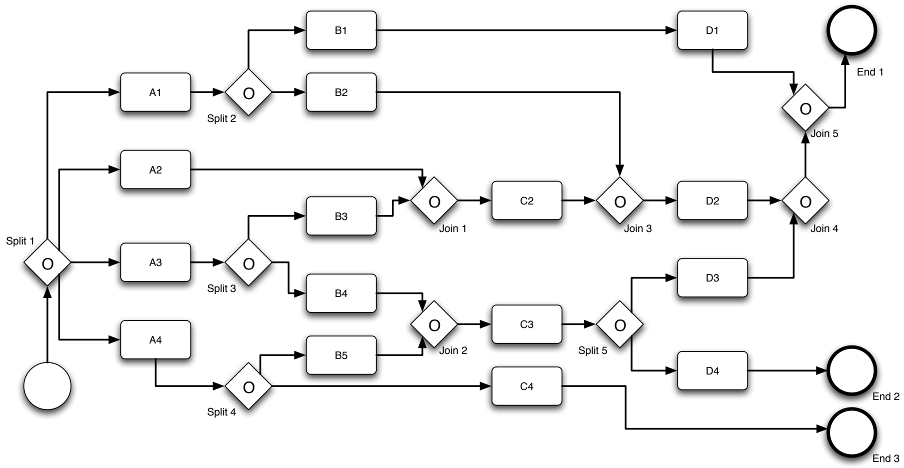


Figure 2: *Acyclic OR splits and joins*

4.2.2 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 $End1$ as in case 1. The additional synchronization tokens produced at $join3$, $join4$, $join5$ on the two paths from $A1$ to $End1$ have three effects. They prevent the rule at $join3$ 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 $join3$ as well as from the arcs entering $join4$ and $join5$ on the path from $B2$ to $End1$), 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 $join5$ two potential synchronizations are required when a token leaves $split1$ to enter $A1$, one on the arc exiting $D1$ and one on the arc exiting $join4$. The first of these two synchronization requests holds in case 2.1 until the token produced upon exiting $split2$ to enter $B1$ arrives at $join5$, 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 $join4$ still no synchronization is necessary since the synchronization tokens produced there between exiting $split1$ and entering $join3$ are deleted upon entering $join3$ 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 [Wynn, 2006], [Wynn et al., 2006b] 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*.

4.2.3 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 $syncToken(node)$ whenever a runtime choice made in a transition upstream *node* excludes a path. Therefore $CtlCondSync(node, select_{Consume}(node))$ 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 *AllJoinArc* function to a function

$NextJoinArc(o)$ 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 $CtlCondSync(node, I)$ to describe synchronization schemes with timeout conditions or similar runtime features.

The very special interpretation of OR-joins by the Synchronizing Merge pattern [van der Aalst et al., 2003] needs neither synchronization tokens nor a $CtlCondSync$, since every token on any single incoming arc is enough to fire the rule. To describe this as an instance of the OrJoinGateTransition($node$) it suffices to refine $select_{Consume}(node)$ to yield singleton sets.

4.2.4 Occam-like OR-join semantics

An OR-join semantics in the style of the parallel programming language Occam and its Transputer implementation [Graham, 1990], [INMOS, 1989] has for each split $node$ a well-defined synchronization node $sync(node)$ where all the processes triggered by a token t at $node$ are synchronized before one can proceed with the next task after $sync(node)$. In particular, $select_{Consume}(sync(node)) = inArc(sync(node))$ holds. 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 $sync(node)$ is known at design time, the production of synchronization tokens is reduced to send from a split $node$ each produced token $t.o$ to its corresponding synchronization arc $sync(o)$; 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. [BPMI.org, 2006, 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 [BPMI.org, 2006, 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.

4.3.1 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 $tokenSet(t)$. We assume that upon a start event a token set $tokenSet(t)$ is generated for the start token t. When new tokens t' appear during the computation, their $tokenSet(t')$ 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 *Enabled*ness to guarantee that each time only tokens of one token set ts are considered.

$$Enabled(in, ts) = (\mid token(in) \cap ts \mid \geq inQty(in))$$

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

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

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

$\textsc{AndJoinGateTransition}(node) = \textsc{WorkflowTransition}(node)$
where

let $\{in_1, \ldots, in_n\} = inArc(node)$
let $\{t_1, \ldots, t_n\} = firingToken(inArc(node))$
let $ts = tokenSet(t_1)$
$CtlCond(node) =$ **forall** $in \in inArc(node)$ $Enabled(in, ts)$
$\textsc{CtlOp}(node) =$
$\textsc{ConsumeAll}(\{(t_i, in_i) \mid 1 \leq i \leq n\})$
$\textsc{Produce}(andJoinToken(\{t_1, \ldots, t_n\}), out)$
$tokenSet(andJoinToken(\{t_1, \ldots, t_n\})) := ts$
$\textsc{ConsumeSyncAll}(\{(t_i, in) | in \in AllJoinArc(out), 1 \leq i \leq n\}$
$\cup \{(t_i, in_i) \mid 1 \leq i \leq n\})$
$\textsc{ProduceSyncAll}$
$(\{(andJoinToken(t_1, \ldots, t_n), in) | in \in AllJoinArc(out)\})$

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:

$CtlCondSync(node, I, ts) =$
forall $i \in I$ $syncToken(i) \cap ts \neq \emptyset$ **and**
forall $i \in inArc(node) \setminus I$ $syncToken(i) \cap ts = \emptyset$

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:

$\textsc{OrJoinGateTransition}(node) = \textsc{WorkflowTransition}(node)$
where
let $I = \{in_1, \ldots, in_n\} = select_{Consume}(node)$
let $\{t_1, \ldots, t_n\} = firingToken(I)$
let $ts = tokenSet(t_1)$ **in**
$CtlCond(node) = (I \neq \emptyset$ **and forall** $i \in I$ $Enabled(i, ts)$ **and** $CtlCondSync(node, I, ts))$
$\textsc{CtlOp}(node) =$
$\textsc{Produce}(orJoinToken(firingToken(I)), out)$
$tokenSet(orJoinToken(firingToken(I))) := ts$
$\textsc{ConsumeAll}(\{(t_i, in_i) \mid 1 \leq i \leq n\})$
$\textsc{ConsumeSyncAll}(\{(t_i, in) | in \in AllJoinArc(out), 1 \leq i \leq n\}$
$\cup \{(t_i, in_i) \mid 1 \leq i \leq n\})$
$\textsc{ProduceSyncAll}$
$(\{(orJoinToken(t_1, \ldots, t_n), in) | in \in AllJoinArc(out)\})$

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.

4.3.2 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 [Sörensen, 2008], 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:

$$cycle(a) :\Leftrightarrow source(a) \in succ^{+}(target(a))$$

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 *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.

OrSplitGateTransition($node$) = WorkflowTransition($node$)
where
  let $\{in\} = inArc(node)$

  let $O = select_{Produce}(node)$
  let $t = firingToken(\{in\})$
    CtlCond($node$) = $Enabled(in, tokenSet(t))$
    CtlOp($node, O$) =
      Consume(t, in)
      ConsumeSyncAll($\{(t, i) \mid i \in AllJoinArc(o)$
        forsome $o \in outArc(node)\}$)
      ProduceSyncAll
        ($\{(orSplitToken(t, o), i) \mid i \in AllJoinArc(o), o \in O\}$)
      forall $o \in O$
        if $cycle(o)$
          Produce($orSplitToken(t, o), o$)
          $tokenSet(orSplitToken(t, o)) := genTokenset(t, o)$
        else
          Produce($orSplitToken(t, o), o$)
          $tokenSet(orSplitToken(t, o)) := tokenSet(t)$

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, *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 *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 $AllJoinArc(o)$ is refined to refer to exactly those incoming edges of joins in the workflow that are reachable from $target(o)$ via a path that contains no outgoing cyclic edge of a split.

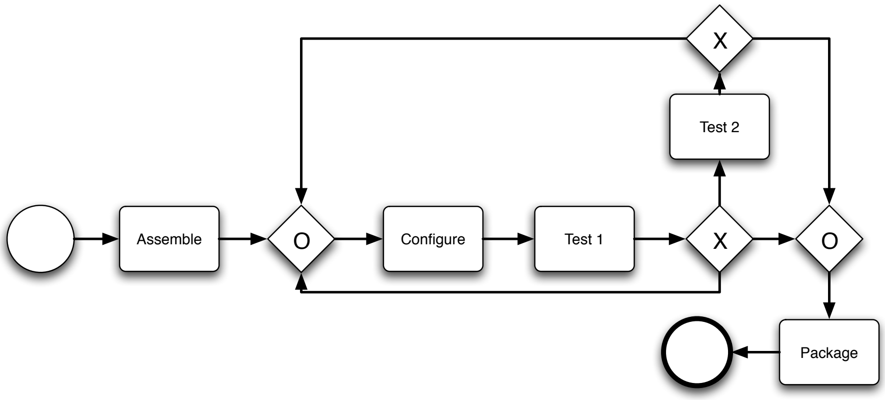


Figure 3: *Production Example from the BPMN Standard*

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 [BPMI.org, 2006] 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” [BPMI.org, 2006, 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 [Sörensen, 2008], 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 is made 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 [Sörensen, 2008] 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 [Batory and Börger, 2008] and choosing appropriate tools to support theorem proving, model checking, static analysis etc.

Acknowledgements

We thank in particular one reviewer for his exceptionally insightful and detailed criticism of the original submission.

References

[A.-W.Scheer, 1994] A.-W.Scheer (1994). *Business Process Engineering: Reference Models for Industrial Enterprises.* Springer-Verlag, New York.

[Batory and Börger, 2008] Batory, D. and Börger, E. (2008). Modularizing theorems for software product lines: The Jbook case study. *J. Universal Computer Science*, 14(12):2059–2082. Extended abstract "Coupling Design and Verification in Software Product Lines" in: S. Hartmann and G. Kern-Isberner (Eds): FoIKS 2008 (Proc. of *The Fifth International Symposium on Foundations of Information and Knowledge Systems*), Springer LNCS 4932, p.1–4, 2008.

[Börger, 2003] Börger, E. (2003). The ASM refinement method. *Formal Aspects of Computing*, 15:237–257.

[Börger, 2007a] Börger, E. (2007a). Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. *Formal Aspects of Computing*, 19:225–241.

[Börger, 2007b] Börger, E. (2007b). Modeling workflow patterns from first principles. In Parent, C., Schewe, K.-D., Storey, V., and Thalheim, B., editors, *Conceptual Modeling–ER 2007*, volume 4801 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag.

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

[Börger and Thalheim, 2008] Börger, E. and Thalheim, B. (2008). A method for verifiable and validatable business process modeling. In Börger, E. and Cisternino, A., editors, *Advances in Software Engineering*, volume 5316 of *LNCS*, pages 59–115. Springer-Verlag.

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

[Dijkman et al., 2007] Dijkman, R. M., Dumas, M., and Ouyang, C. (2007). Formal semantics and automated analysis of BPMN process models. Technical Report 5969, Queensland University of Technology, Brisbane.

[Dumas et al., 2007] Dumas, M., Grosskopf, A., Hettel, T., and Wynn, M. (2007). Semantics of BPMN process models with or-joins. In Meersman, R. and et al., Z. T., editors, *OTM 2007 Part I*, volume 4803 of *Lecture Notes in Computer Science*, pages 41–58. Springer.

[Graham, 1990] Graham, I. (1990). *The Transputer Handbook*. Prentice-Hall.

[Grosskopf, 2007] Grosskopf, A. (2007). xBPMN. Formal control flow specification of a BPMN based process execution language. Master's thesis, HPI at Universität Potsdam. pages 1-142.

[Gruhn and Laue, 2005] Gruhn, V. and Laue, R. (2005). Einfache EPK-Semantik durch praxistaugliche Stilregeln. In *Geschäftsprozessmanagement mit ereignisgesteuerten Prozessketten*, pages 176–189.

[Gruhn and Laue, 2006] Gruhn, V. and Laue, R. (2006). How style checking can improve business process models. In *Proc. 8th International Conference on Enterprise Information Systems (ICEIS 2006)*, Paphos (Cyprus).

[Gruhn and Laue, 2007] Gruhn, V. and Laue, R. (2007). What business process modelers can learn from programmers. *Science of Computer Programming*, 65:4–13.

[INMOS, 1989] INMOS (1989). *Transputer Implementation of Occam – Communication Process Architecture*. Prentice-Hall, Englewood Cliffs, NJ.

[Kindler, 2004] Kindler, E. (2004). On the semantics of EPCs: A framework for resolving the vicious circle. In J.Desel, Pernici, B., and M.Weske, editors, *Proceedings of 2nd International Conference on Business Process Management*, volume 3080 of *LNCS*, pages 82–97. Springer-Verlag.

[Kindler, 2005] Kindler, E. (2005). On the semantics of EPCs: resolving the vicious circle. *Data and Knowledge Engineering*, 56:23–40.

[Mendling et al., 2006] Mendling, J., Moser, M., Neumann, G., Verbeek, H., Dongen, B., and van der Aalst, W. (2006). A quantitative analysis of faulty EPCs in the SAP reference model. Technical Report BPM-06-08, BPMcenter.org.

[Rittgen, 1999] Rittgen, P. (1999). Modified EPCs and their formal semantics. Technical Report 99/19, Institut für Informationssyteme, Universität Koblenz-Landau.

[Russel et al., 2007a] Russel, N., ter Hofstede, A., van der Aalst, W. M. P., and Edmond, D. (2007a). *new*yAWL: Achieving comprehensive patterns support in workflow for the contro-flow, data and resource perspectives. BPM-07-05 at BPMcenter.org.

[Russel et al., 2006] Russel, N., ter Hofstede, A., van der Aalst, W. M. P., and Mulyar, N. (2006). Workflow control-flow patterns: a revised view. BPM-06-22 at http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/.

[Russel et al., 2007b] Russel, N., ter Hofstede, A. H. M., and van der Aalst, W. M. P. (2007b). *new*YAWL: Specifying a workflow reference language using coloured Petri nets. Eighth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus, Denmark.

[Sörensen, 2008] Sörensen, O. (2008). Semantics of joins in cyclic BPMN workflows. Master's thesis, University of Kiel. www.is.informatik.uni-kiel/˜ thalheim/ASM/MetaProgrammingASM.

[van der Aalst,] van der Aalst, W. Pi calculus versus Petri nets: Let us eat "humble pie" rather than inflate the "Pi hype". downloaded February 2008. http://is.tm.tue.nl/research/patterns/download/pi-hype.pdf.

[van der Aalst et al., 2002] van der Aalst, W., Desel, J., and Kindler, E. (2002). On the semantics of EPCs: A vicious circle. In Rump, M. and Nüttgens, F. G., editors, *Proc. of the EPK 2002: Business Process Management using EPCs*, pages 71–80, Trier. Gesellschaft für Informatik.

[van der Aalst and ter Hofstede, 2005] van der Aalst, W. and ter Hofstede, A. (2005). YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275.

[van der Aalst et al., 2003] van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., and Barros, A. (2003). Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51.

[van Hee et al., 2006] van Hee, K., Oanea, O., Serebrenik, A., Sidorova, N., and Voorhoeve, M. (2006). History-based joins: Semantics, soundness and implementation. In Dustdar, S., Fiadeiro, J. L., and Seth, A., editors, *BPM 2006*, volume 4102 of *LNCS*, pages 225–240. Springer-Verlag.

[Wynn, 2006] Wynn, M. (2006). *Semantics, Verification, and Implementation of Workflows with Cancellation Regions and OR-Joins.* PhD thesis, Queensland University of Technology.

[Wynn et al., 2005] Wynn, M., Edmond, D., van der Aalst, W., and ter Hofstede, A. (2005). Achieving a general, formal and decidable approach to the

OR-join in workflow using reset nets. In *Application and Theory of Petri Nets 2005*, volume 3536 of *LNCS*, pages 423–443. Springer.

[Wynn et al., 2006a] Wynn, M., van der Aalst, W., ter Hofstede, A., and Edmond, D. (2006a). Synchronisation and cancellation in workflows based on reset nets. Technical Report BPM-06-26, BPMcenter.org.

[Wynn et al., 2006b] Wynn, M., van der Aalst, W., ter Hofstede, A., and Edmond, D. (2006b). Verifying workflows with cancellation regions and OR-joins: an approach based on reset nets and reachability analysis. In Dustdar, S., Fiadeiro, J. L., and Seth, A. P., editors, *Business Process management BPM 2006*, volume 4102 of *LNCS*, pages 389–394. Springer-Verlag. Previous versions edited as BPM-06-16 and BPM-06-12.

[Wynn et al., 2006c] Wynn, M., Verbeek, H. M. W., van der Aalst, W., ter Hofstede, A., and Edmond, D. (2006c). Reduction rules for reset workflow nets. Technical Report BPM-06-25, BPMcenter.org.

[Wynn et al., 2006d] Wynn, M., Verbeek, H. M. W., van der Aalst, W., ter Hofstede, A., and Edmond, D. (2006d). Reduction rules for YAWL workflow nets with cancellation regions and OR-joins. Technical Report BPM-06-24, BPMcenter.org.