Semantics of Standard Process Models with OR-Joins

Marlon Dumas[1,2], Alexander Grosskopf[3], Thomas Hettel[4,1], and Moe Wynn[1]

[1] Queensland University of Technology, Australia
{m.dumas, m.wynn}@qut.edu.au
[2] University of Tartu, Estonia
marlon.dumas@ut.ee
[3] Hasso-Plattner Institute, Potsdam, Germany
alexander.grosskopf@hpi.uni-potsdam.de
[4] SAP Research Centre Brisbane, Australia
t.hettel@sap.com

Abstract. The Business Process Modeling Notation (BPMN) is an emerging standard for capturing business processes. Like its predecessors, BPMN lacks a formal semantics and many of its features are subject to interpretation. One construct of BPMN that has an ambiguous semantics is the OR-join. Several formal semantics of this construct have been proposed for similar languages such as EPCs and YAWL. However, these existing semantics are computationally expensive. This paper formulates a semantics of the OR-join in BPMN for which enablement of an OR-join in a process model can be evaluated in quadratic time in terms of the total number of elements in the model. This complexity can be reduced down to linear-time after materializing a quadratic-sized data structure at design-time. The paper also shows how to efficiently detect the enablement of an OR-join incrementally as the execution of a process instance unfolds.

1 Introduction

Business process management as a discipline has traditionally suffered from a proliferation of process definition languages based on similar but subtly different concepts and constructs. After numerous attempts, standardization efforts in this space have converged towards two languages: the Business Process Modeling Notation (BPMN) [13], which is intended for modeling business processes primarily during the analysis and design phases, and the Business Process Execution Language (BPEL) [9], which is intended for implementation and execution of business processes in a service-oriented architecture. The standard specifications of both of these languages are given in a narrative, informal style. In the case of BPEL, a number of formalizations have been proposed [3]. On the other hand, virtually no attempt has been made to attach a formal semantics to BPMN, barring recent work on formalizing subsets thereof [17,7]. Compounded with the fact that executability has not been a major concern during the standardization of BPMN, this has led to a standard specification with
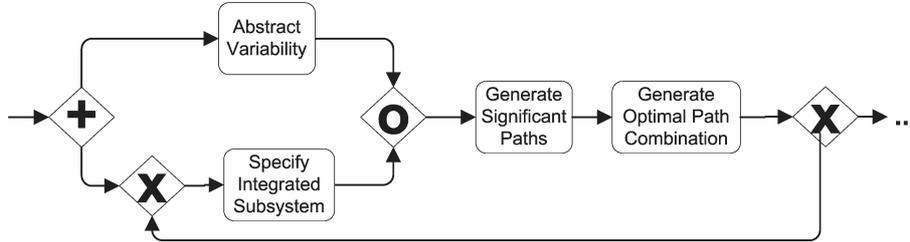
Fig. 1. Process fragment with an OR-join (example inspired from [14])

numerous ambiguities. This situation raises the risk that different tools adopt different interpretations of BPMN, especially those tools supporting process simulation and automated generation of executable process definitions which start to emerge [16].

In separate work, we formalized a subset of BPMN [7]. One of the constructs left aside in this formalization, as well as in reference [17], is the *inclusive merge gateway* (hereafter called the *OR-join*). This construct corresponds to a workflow pattern called "Synchronizing Merge" [2]. An OR-join is a point in a process where several branches converge. For each of its incoming branch, the OR-join will normally wait for a token indicating its completion; but if at some point in time it can be determined that no token will ever arrive along a given incoming branch, the OR-join will not wait for a token along that branch.

Figure 1 shows a use case of the OR-join. The first time this process fragment is executed, both tasks "Abstract Variability" and "Specify Integrated Subsystem" are executed in parallel. This parallel execution is captured by the AND-split gateway (a lozenge with a "+" symbol). The OR-join (lozenge with an "O" symbol) will then wait for both tasks to complete. Thereafter, the execution proceeds along task "Generate Significant Paths" followed by "Generate Optimal Path Combination". After completion of this latter task, a choice is made between repeating task "Specify Integrated Subsystem" or proceeding with the rest of the process (not shown in the figure). When the second execution of "Specify Integrated Subsystem" completes, the OR-join receives a token along one of its incoming branches. The OR-join can fire at this point without waiting for a token from task "Abstract Variability", because this task is not executed in the second round. In other words, the branch coming from task "Abstract Variability" into the OR-join is not "active", and thus the OR-join will not wait for a token along this branch.

Formalizing the OR-join is challenging as highlighted by previous experiences in defining semantics for languages that incorporate this construct, such as EPCs [10] and YAWL [19]. Unlike other routing constructs, the OR-join has a non-local semantics: in order to determine whether or not an OR-join is enabled, it is not sufficient to examine the presence of tokens in its immediate vicinity. Instead, enablement of an OR-join may depend on the presence or absence of
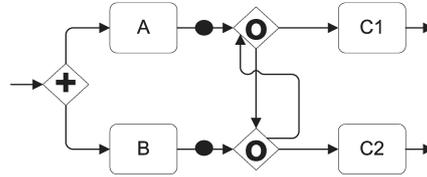
Fig. 2. Example of a vicious circle. Tokens are represented as black dots.

tokens in places far away in the model. Not surprisingly, the evaluation of enablement for OR-joins using existing semantics is computationally expensive.

Another issue when defining a semantics for the OR-join, is that the state of enablement of an OR-join (i.e. its ability to fire at a given point during the execution) may depend on the state of enablement of another OR-join in the model and vice-versa. In other words, two OR-joins may end up waiting for one another, a situation known as a *vicious circle* [1]. An example of a vicious circle is given in Figure 2. One may argue that such scenarios are hypothetical and there is no harm in excluding them or giving them an arbitrary semantics that generates deadlocks. However, in this paper we show that, without adding to the complexity, we can define a semantics for the OR-join in BPMN that is able to deal with such scenarios without generating deadlocks.

The contribution of this paper is a semantics of the OR-join in BPMN that strikes a balance between precision in determining when an OR-join should fire, and the computational complexity of determining whether or not an OR-join is enabled. After formulating this semantics, the paper presents an algorithm that allows enablement of a given OR-join in a model to be determined in quadratic time in terms of the total number of elements in the model. This complexity can be reduced to linear time if a quadratic-sized data structure is materialized at design-time for each OR-join in the model. Furthermore, we show that the proposed algorithm can be adapted to an incremental evaluation mode.

The rest of the paper is structured as follows. Some background on BPMN is given in Section 2. Next, the semantics of BPMN models with OR-joins is presented in Section 3, while Section 4 describes an algorithm for evaluating enablement of an OR-join in a given state. Section 5 introduces an incremental version of the algorithm that provides some additional optimization. Finally, Section 6 discusses related work while Section 7 concludes.

2   Syntax of BPMN

This section introduces the BPMN notation and provides an abstract syntax for BPMN process models.

2.1   BPMN Overview

A process model in BPMN is represented as a *Business Process Diagram* (BPD). A BPD is a graph in which nodes correspond to activities or routing steps
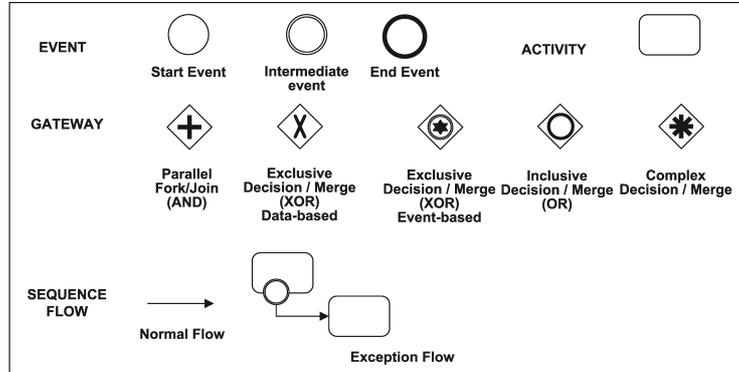
Fig. 3. BPMN graphical elements

(collectively called *objects*), while edges correspond to flows of control or flows of messages. Objects and flows can be grouped into pools and swimlanes to capture domains of responsibility. They can also be associated with artifacts that capture non-functional information. In this paper, we concentrate on the control-flow semantics of BPMN and thus we leave aside pools, swimlanes and artifacts. Also, since we only consider the semantics of one process model at a time, as opposed to the semantics of multiple communicating processes, we leave aside message flows. Figure 3 summarizes the constructs we focus upon.

An object can be an *event*, an *activity* or a *gateway*. An event, depicted as a circle, represents something that can affect a process execution. Events are classified based on their position in the graph into start events (events that are source in the graph), end events (events that are sink in the graph) and intermediate events. Events are also classified based on their triggering cause into timer events, message events, etc. Given that the cause of an event is not relevant from a control-flow perspective, we do not consider this latter classification.

An *activity*, depicted as a rounded rectangle, represents a unit of work. An activity may correspond to an undecomposed task or to a subprocess invocation. In this paper, we capture the execution of one individual process at a time. If this process invokes another one, the invoked subprocess is seen as a black-box, and accordingly, we treat its execution as that of an undecomposed task.

A *gateway* is used to control branching, forking, merging, and joining of paths and is represented using a diamond. There are different gateway types: (i) *exclusive gateways* for selecting one branch among a set of alternative branches based on data or events (XOR-split), or for merging a number of alternative branches (XOR-join); (ii) *parallel gateways* for forking one branch into multiple concurrent branches (AND-split) or for merging multiple concurrent branches into one (AND-join) using a barrier-synchronization policy; (iii) *inclusive gateways* for choosing one or multiple branches based on boolean expressions (OR-split), or for synchronizing multiple branches while waiting only for those branches that

are active (OR-join); and (iv) *complex gateway* for modeling complex branching conditions and synchronization policies (e.g., wait for 3 branches out of 5).

Objects in a BPD are related by means of *sequence flows* and *exception flows*. A sequence flow captures a sequential dependency: when the task completes normally, a token is placed on each of its outgoing sequence flows. Meanwhile, if an error occurs during the performance of an activity, the activity is interrupted and a token is placed in the exception flow corresponding to that error.

2.2 Abstract Syntax

Abstracting from its concrete syntax, a BPD can be thought of as containing various types of objects and flows as captured by the following definition.

Definition 1 (Business Process Diagram (BPD)). *A BPD is a tuple* $\mathcal{BPD}=(\mathcal{O}, \mathcal{A}, \mathcal{G}, \mathcal{E}, \mathcal{G}^{\mathcal{X}}, \mathcal{G}^{\mathcal{P}}, \mathcal{G}^{\mathcal{I}}, \mathcal{G}^{\mathcal{C}}, \mathcal{G}^{\mathcal{E}}, \mathcal{E}^{\mathcal{S}}, \mathcal{E}^{\mathcal{I}}, \mathcal{E}^{\mathcal{E}}, \mathcal{F}, abf)$, where*

- *$\mathcal{O}$ is a set of objects which can be partitioned into disjoint sets of activities $\mathcal{A}$, gateways $\mathcal{G}$ and events $\mathcal{E}$,*
- *$\mathcal{G}$ is a set of gateways which can be partitioned into disjoint sets of exclusive OR gateways $\mathcal{G}^{\mathcal{X}}$, parallel AND gateways $\mathcal{G}^{\mathcal{P}}$, inclusive OR gateways $\mathcal{G}^{\mathcal{I}}$, complex gateways $\mathcal{G}^{\mathcal{C}}$ and event-based gateways $\mathcal{G}^{\mathcal{E}}$,*
- *$\mathcal{E}$ is a set of events which can be partitioned into disjoint sets of start events $\mathcal{E}^{\mathcal{S}}$, intermediate events $\mathcal{E}^{\mathcal{I}}$ and end events $\mathcal{E}^{\mathcal{E}}$,*
- *$\mathcal{F} \subseteq \mathcal{O} \times \mathcal{O}$ is the flow relation between objects.*
- *$abf : \mathcal{G}^{\mathcal{C}} \rightarrow \mathcal{O}$ is a function capturing the preconditions for a complex gateway to be enabled, as discussed later in Section 3.2.*

The relation $\mathcal{F}$ defines a directed graph over the set of objects $\mathcal{O}$. For any object $x \in \mathcal{O}$, the set of direct predecessors is given by $pred(x) = \{y \in \mathcal{O} | y\mathcal{F}x\}$ and analogously the set of direct successors is given by $succ(x) = \{y \in \mathcal{O} | x\mathcal{F}y\}$. $\mathcal{F}^*$ is the reflexive transitive closure of $\mathcal{F}$. The set of all direct and transitive predecessors of an object x is given by function $pred^*(x) = \{y \in \mathcal{O} | y\mathcal{F}^*x\}$.

A BPD as defined in Definition 1 has no structural requirements in terms of a starting point or an end point. Typically, a business process model has one starting point, one or more end points and all the objects used in the modeled are connected. Definition 2 defines minimal structural requirements for a (connected) BPD, i.e., there is a start event, there is one or more end events and every object is on a path from the start event to an end event.

Definition 2 (Connected BPD). *A BPD = ($\mathcal{O}$, $\mathcal{A}$, ...) is connected if it satisfies the following conditions:*

- *there is exactly one start event with zero incoming flow and at least one outgoing flow: $|\mathcal{E}^{\mathcal{S}}| = 1 \wedge \exists s \in \mathcal{E}^{\mathcal{S}} pred(s) = \emptyset \wedge |succ(s)| \geq 1$*
- *there is one or more end events with incoming flows and zero outgoing flow: $|\mathcal{E}^{\mathcal{E}}| \geq 1 \wedge \forall e \in \mathcal{E}^{\mathcal{E}} |pred(e)| \geq 1 \wedge succ(e) = \emptyset$*
- *every object (other than the start and end events) is on a path from a start event to an end event: $\forall x \in \mathcal{O} \setminus (\mathcal{E}^{\mathcal{S}} \cup \mathcal{E}^{\mathcal{E}}) \exists s \in \mathcal{E}^{\mathcal{S}} \exists e \in \mathcal{E}^{\mathcal{E}} s\mathcal{F}^*x \wedge x\mathcal{F}^*e.*

We assume that all BPDs are connected. We also assume, without loss of generality, that every gateway in a BPD is either a split gateway (it has one incoming flow and multiple outgoing flows) or a join gateway (multiple incoming flows and one outgoing flow). Similarly, we assume that activities and intermediate events have only one incoming flow and one outgoing flow. It is straightforward to expand a BPD that does not satisfy these conditions into one that does by applying simple expansion rules. For example, if an activity has multiple incoming flows it is equivalent to a structure where these flows lead to an XOR-join gateway and this gateway has a flow that leads into the activity in question. The above conditions together with the connectedness requirement entail that no activity or event in a BPD is both the source and target of the same flow.

We have excluded exception flows from the abstract syntax. To simplify the presentation of the proposed OR-join semantics and without loss of generality, we assume that a BPD is pre-processed as follows in order to replace all exception flows with gateways and sequence flows. If an activity contains at least one exception flow, all the outgoing flows of this activity are deleted and replaced by one single sequence flow that leads to an exclusive decision gateway (i.e. an XOR-split). This XOR-split has multiple branches: one branch corresponds to the sequence flow going out of the activity, and the other branches correspond to the various exception flows, and these exception flows are replaced with normal sequence flows. The idea is that once an activity completes (whether normally or abnormally) a decision is made to determine if the sequence flow is taken (if the task completed normally), or one of the exception flows is taken (if the task completed due to an error). This decision is captured by such a decision gateway, and in doing so, all the exception flows are replaced with sequence flows. Hereafter, we will use the term *flow* to refer to a *sequence flow*.

3   Semantics of BPMN Models

This section formulates a semantics for the OR-join in BPMN. After some considerations regarding the notion of the OR-join, the section introduces a formal definition of enablement of objects in BPMN process models with OR-joins.

3.1   Informal Definition

The BPMN specification makes vague statements about the meaning of the OR-join, such as: "the Inclusive Gateway [...] will wait for (synchronize) all Tokens that have been produced upstream. It does not require that all incoming Sequence Flow produce a Token (as the Parallel Gateway does). It requires that all [Tokens] that were actually produced by an upstream [object arrive]."[1] The clarity of this statement is hampered by the fact that the term "upstream" is not defined, and its meaning is unclear if there are cycles in the graph. Also, this statement does not clarify how many tokens from each incoming flow should the OR-join wait for.

---

[1] This sentence is incomplete in the specification, so we have added the last two words.

From a detailed reading of the BPMN specification, and from the definition of the Synchronizing Merge pattern [2] to which the BPMN specification refers to, we can distill the following characteristics of the OR-join:

- In line with the definition of all other gateways in BPMN, a necessary condition for an OR-join to be enabled is that there is at least one token in at least one of its incoming flows.
- A sufficient condition for an OR-join to be enabled is that there is at least one token in each of its incoming flows, i.e. all branches have "completed".
- If an incoming flow of the OR-join has no token, a necessary condition for the OR-join to be enabled is that it is not possible for a token to reach this flow. This captures the notion of waiting for tokens produced "upstream".

Thus an OR-join has a behavior in-between the XOR-join and the AND-join. The XOR-join waits for one token in one of its incoming flows, while the AND-join waits for one token in each of its incoming flows. The OR-join may behave like an XOR-join, or like an AND-join, or like something in-between depending on the state of the process instance. Another characteristic of the OR-join is that it only waits for tokens that will eventually arrive. As a result, an OR-join will not deadlock in situations where an AND-join would. For example, if we replaced the OR-join in Figure 1 with an AND-join, a deadlock would occur if task "Specify Integrated Subsystem" was repeated.

We decompose the control-flow semantics of objects in BPMN into an *enablement rule* and a *firing rule*. The enablement rule determines if the object is ready to fire in a given state. If one or more objects are enabled, the execution environment may select any one of them and fire it. The firing rule determines what happens then. For example, when an AND-join fires, it consumes one token from each of its incoming flows and it produces one token in its outgoing flow. On the other hand, when an OR-join fires, it consumes one token from each incoming flow that has a token, and it produces a token in the outgoing flow. The definition of firing rules for the various types of BPMN objects, including gateways, does not pose major challenges. Accordingly, we focus on the enablement rules, especially the one for the OR-join.

Given the above characteristic of the OR-join, we propose the following OR-join enablement rule:

> An OR-join o is enabled if there is at least one token in one of its incoming flows, and for each of its incoming flows f, either f has at least one token or, taking as an assumption that o will not fire, no token will arrive to flow f through a sequence of firings starting from the current state.

One variable in this informal definition is how to determine that no token will ever arrive to flow f. In the semantics of the OR-join for YAWL [18], a far-sighted approach is taken. Conceptually, the entire set of possible future states is computed to determine if, in any of these possible states, there will be at least one token in the incoming flow in question. This far-sighted approach ensures that the OR-join is enabled as early as possible. But its computational complexity

is proportional to the number of possible states, as opposed to the number of elements in the model. To avoid this computational problem, we propose a "myopic" (or "short-sighted") approach: we can determine that no token will ever arrive to flow f, if we can determine that none of the direct or indirect predecessors of f is enabled.

This definition also needs some refinement to deal with cycles containing OR-joins. Hereafter, we say that two OR-joins are in *structural conflict* if one is a predecessor of the other and vice-versa. At a given state, two OR-joins are said to be in *partial conflict* if they are in structural conflict and each of them has at least one token in at least one of its incoming flows. Finally, two OR-joins are said to be in *full conflict* iff they are in structural conflict and they are only waiting for each other to fire first so that they can become enabled. This scenario was illustrated in Figure 2. Also, if an OR-join is part of a loop, this OR-join is in structural conflict with itself, and if some of its input flows contain a token, the OR-join may end up being in full conflict with itself.

With respect to the above enablement rule, partial and full conflicts raise the following issue: In the process of determining whether the OR-join o_1 is enabled, we may need to recursively ask ourselves the question of whether o_1 is enabled; or in the process of determining whether o_1 is enabled, we ask the question of whether another OR-join o_1' is enabled, and in determining whether o_1' is enabled we ask the question of whether o_1 is enabled. At least two approaches are possible to break such vicious circles [19]. We could be "optimistic" and say that o_1 is enabled if it has at least one of its input flow marked. Or we can be "pessimistic" and say that o_1 is enabled if and only if tokens are present at each of its incoming branches. Adopting an optimistic strategy can lead to deadlocks [20], for example in the full conflict depicted in Figure 2. Accordingly, in this paper, we adopt a pessimistic strategy: in the case of a partial or full conflict between o_1 and o_1', gateway o_1 will treat o_1' as not enabled, and reciprocally, o_1' will treat o_1 as not enabled. Consequently, both o_1 and o_1' will be enabled if they are in full conflict.

3.2   Formal Definition of Enablement

From a control-flow perspective, the state of an instance of a BPD can be captured as a set of tokens distributed among the flows composing the BPD. We can think of a flow $f \in \mathcal{F}$ as a place where tokens are stored. Initially, one token is located in each of the flows emanating from the start event of a BPD. As the execution of the BPD proceeds, tokens are removed from certain flows and added to other flows according to the firing rules. Hence, the state of a process instance at a given state can be captured using a token count function $tc : \mathcal{F} \to \mathbb{N}$ that takes as input a flow and returns the number of tokens stored in that flow. Flows are represented as pairs of objects (o_1, o_2).

Definition 3 formalizes the conditions for enablement of different types of objects. It defines a function that takes as parameter an object, a state of a BPD execution and a set of "visited objects" V.

Definition 3 (Object Enablement). *Given a BPD and a token count function* tc, *an object o is enabled in the state represented by* tc *iff* enabled(o, tc, ∅) *is true, where:* enabled: $\mathcal{O} \times (\mathcal{F} \rightarrow \mathbb{N}) \times (\wp(\mathcal{O}) \rightarrow \mathbb{B})$ *is defined as follows:*

$enabled(o_1, tc, V) =$
$\quad o_1 \in \mathcal{G}^{\mathcal{X}} \wedge \exists o_2 \in pred(o_1) tc((o_2, o_1)) \geq 1 \vee$
$\quad o_1 \in (\mathcal{A} \cup \mathcal{E}^{\mathcal{I}}) \wedge \exists o_2 \in pred(o_1) \setminus \mathcal{G}^{\mathcal{E}} tc((o_2, o_1)) \geq 1 \wedge$
$\qquad\qquad \exists o_2 \in pred(o_1) \cap \mathcal{G}^{\mathcal{E}} \exists o_3 \in pred(o_2) tc((o_3, o_2)) \geq 1 \vee$
$\quad o_1 \in \mathcal{G}^{\mathcal{P}} \wedge \forall o_2 \in pred(o_1) tc((o_2, o_1)) \geq 1 \vee$
$\quad o_1 \in \mathcal{G}^{\mathcal{C}} \wedge \exists s \in abf(o_1) \forall o_2 \in s\ tc(o_2, o_1) \geq 1 \vee$
$\quad o_1 \in \mathcal{G}^{\mathcal{I}} \wedge o_1 \notin V \wedge \exists o_2 \in pred(o_1) tc(o_2, o_1) \geq 1 \wedge$
$\qquad\qquad \forall o_2 \in pred(o_1) tc(o_2, o_1) = 0 \Rightarrow$
$\qquad\qquad\qquad \neg\exists o_3 \in pred^*(o_2) enabled(o_3, tc, V \cup \{o_1\})$

For objects other than OR-joins, it is straightforward to determine if they are enabled. For example, an exclusive gateway ($\mathcal{G}^{\mathcal{X}}$) is enabled if there is at least one token in at least one of its incoming flows.[2] Activities ($\mathcal{A}$) and intermediate events ($\mathcal{E}^{\mathcal{I}}$) are also enabled if there is at least one token in their incoming flow. An exception to this latter rule occurs if one of the predecessor of the activity or event in question is an event-driven choice gateway. In this case, the activity/event is enabled if there is at least one token in one of the incoming flows of that event-driven choice gateway. A parallel gateway ($\mathcal{G}^{\mathcal{P}}$) requires all incoming flows to carry a token for enablement. For the complex gateway ($\mathcal{G}^{\mathcal{C}}$), a subset of the incoming flows need to all contain at least one token. In the concrete syntax of BPMN, a boolean condition over sequence flows is given to capture under which situations is a complex gateway enabled, i.e. which are the possible subsets of incoming flows that need to contain tokens for the complex gateway to fire. Here, we abstract away from the concrete syntax and we assume the existence of a function *abf* which given a complex gateway o_1 in a BPD, retrieves the set of possible subsets of predecessors of o_2, such that a token needs to be present in each flow (o_2, o_1) for the complex gateway to be enabled.

For inclusive OR Gateways ($\mathcal{G}^{\mathcal{I}}$), specifically those with more than one incoming flow (i.e. OR-joins), the semantics is more complicated. The informal semantics is such that if any object in the set of predecessors of an OR-join is enabled, the OR-join should wait. If there is at least one token at each of the incoming flows of an OR-join, the OR-join is clearly enabled. Otherwise, it is necessary to explore some or all of the predecessors of the OR-join, to detect whether the OR-join needs to wait for them or not. This may lead to a recursive definition if an OR-join is its own set of predecessors. To avoid an infinite recursion, the definition of enablement keeps track of the set of OR-joins that have been visited. This is the role of parameter V.

The first time an OR-join is visited, its semantics is treated as non-local: the OR-join is enabled if and only if there is at least one token in one of the incoming flows and all predecessors along incoming flows with no tokens are disabled. For

---

[2] Event-driven choice gateways ($\mathcal{G}^{\mathcal{E}}$) are never enabled. Their presence however determines whether or not the objects that immediately succeed them are enabled.
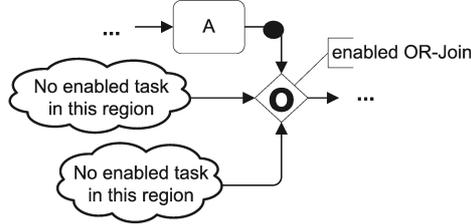
Fig. 4. Checking if a non-visited OR-join is enabled

example, Figure 4 shows an OR-join with three incoming flows. The flow coming from A contains a token (shown as a black dot). To determine if this OR-join is enabled, we inspect the set of predecessors along the other flows (which have no tokens in the current state) and we check that none of them is enabled.

Meanwhile, if an OR-join has already been visited, it is necessarily in partial or full conflict with another OR-join. Indeed, an object o is only added to the set of visited objects V by a recursive call to function *enabled*, with the first parameter of this call being a predecessor of o along an empty flow. So if o has already been visited, it means that there is an OR-join o'_1 such that $o'_1 \in$ pred*$(o_1) \land o_1 \in$ pred*(o'_1) and such that function *enabled* has been previously called with o'_1 as first parameter and this call generated a call to *enabled* with o_1 as first parameter.[3] Thus, we have a situation where, to determine if o'_1 is enabled, we ask the question of whether or not o_1 is enabled, and vice-versa as depicted in Figure 2. To resolve this conflict, we treat o_1 as not enabled with respect to o'_1. Accordingly, if $o_1 \in V$, the function evaluates to false.

4   Algorithm for Determining Enablement of an OR-Join

A naive implementation of the formal definition of enablement is computationally expensive. To determine whether an OR-join is enabled, a naive algorithm needs to potentially visit every predecessor of the OR-join (transitive or not), and for each non-visited OR-join in this set of predecessors, it needs to make a recursive call. In the worst-case, each object is a predecessor of each other and then, the number of recursive calls of the enablement function is equal to the number of objects in the BPD minus the number of visited objects. Thus, the complexity of the naive algorithm is captured by the following recursive function: $c(T, X) = (N - X) \times c(N, X - 1)$ where N is the number of objects in the BPD and X is the number of visited objects. Given a BPD with T tasks, the complexity of the function call enabled(o, tc, { }) is thus in the order $O(T!)$.

The problem with the naive algorithm is that each object is examined a repeated number of times, and each time, the same question is asked, namely "is the object enabled?" If the BPD is primarily composed of OR-joins and all these OR-joins are in structural conflict, this leads to a combinatorial explosion.

---

[3] Here, o_1 and o'_1 may be the same object.

Below we present an algorithm that overcomes this combinatorial explosion by avoiding the recursion. We achieve this by making the following observation: An OR-join for which enablement needs to be determined (say gateway o_1) does not actually need to know whether a preceding OR-join (say o'_1) is enabled or not. What is important is to determine if o_1 must wait for o'_1 assuming that o_1 is not waiting for any other of its precedessors. Indeed, the algorithm will visit all relevant predecessors of o_1, and if it determines that o_1 needs to wait for any of them, the algorithm will return false. Under this assumption gateway o_1 must wait for o'_1 if the following conditions hold:

1. There is at least one token in at least one incoming flow of o'_1.
2. There is at least one token in each incoming flow of o'_1 that is part of a path starting at o_1 and finishing at o'_1.[4]

The first condition is a necessary condition for enablement of an OR-join, and thus it is a necessary condition for o_1 to have to wait for o'_1. The second condition is also necessary. If this condition was false for a given path from o_1 to o'_1, then o_1 and o'_1 are in full conflict, thus entailing that o_1 must not wait for o'_1. Indeed, we are assuming that o_1 is not waiting for any other of its predecessors to fire. We can then conclude that o'_1 is not waiting for any of its predecessors neither (apart from o_1) since all predecessors of o'_1 are also predecessors of o_1. So o_1 and o'_1 are waiting only for one another, and hence they are in full conflict.

The two conditions are also sufficient for o_1 to have to wait for o'_1. Indeed, one of the characteristics of the definition of the OR-join is that if there is a token in one of the incoming flows of this OR-join (o'_1 in this case), this OR-join will eventually fire. Hence, if the first condition is true then o_1 must wait for o'_1 unless there is a possibility of a full conflict between the two gateways. This latter case is excluded if the second condition also holds.

We split the proposed algorithm into two functions: one for determining the enablement of any object except an OR-join, and the other for OR-joins. Accordingly, we first define a function *IsObjectEnabled* that determines if a given object is enabled for a BPD (see Figure 5). This function implicitly takes as input a BPD, but for simplicity, the BPD is not shown as a parameter; instead, the components of the BPD (e.g. $\mathcal{G}^{\mathcal{I}}$) are referred to in the body of the function.

Function call *IsObjectEnabled*(o, tc) returns true iff object o is enabled in state tc. Unlike the enablement function in Section 3.2, *IsObjectEnabled* does not maintain a set of visited objects, as it is not recursive. This function handles all gateways except the OR-join. Enablement of an OR-join is determined by another function, namely *IsOREnabled* – see Figure 6. The function call *IsOREnabled*(o, tc) returns true iff OR-join o is enabled in state tc.

The algorithm first checks that the OR-join o has at least one token in at least one of its incoming flows. If so, it iterates over the set of predecessors of the OR-join along an empty incoming flow (i.e. an incoming flow of o that has no

---

[4] Importantly, if there is no path from o_1 and o'_1, meaning that there is no cycle involving both o_1 and o'_1, this condition is true.

```
1    FUNCTION IsObjectEnabled(o₁ : 𝒪, tc : ℱ → ℕ): 𝔹
2    case
3      o₁ ∈ (𝒢^𝒳 ∪ 𝒢^ℰ):
4        return ∃o₂ ∈ pred(o₁) tc(o₂, o₁) ≥ 1;
5        ... (other cases except o₁ ∈ 𝒢^ℐ treated as per Definition 3)
6      o₁ ∈ 𝒢^ℐ:
7        return IsOREnabled(o₁, tc);
8    end case;
```

Fig. 5. Algorithm to determine whether an object is enabled

```
1    FUNCTION IsOREnabled(o : 𝒢^ℐ, tc : ℱ → ℕ) : 𝔹
2    if ¬∃po ∈ pred(o) tc((po, o)) ≥ 1 then
3      return false;
4    else
5      PredAlongEmptyFlows := {o′ ∈ pred*(po) | ∃po ∈ pred(o) tc((po, o)) = 0}
6      foreach o′ ∈ PredAlongEmptyFlows do
7        if o′ ∈ 𝒢^ℐ ∧ ∃po′ ∈ pred(o′) tc((po′, o′)) ≥ 1 ∧
8          ∀po′ ∈ pred(o′) o ∈ pred*(po′) ⇒ tc((po′, o′)) ≥ 1 then
9          return false
10         else if o′ ∉ 𝒢^ℐ ∧ IsObjectEnabled(o′, tc) then
11           return false;
12         end if
13       end foreach;
14     return true;
```

Fig. 6. Optimized algorithm to determine if an OR-join is enabled.

token). For each of these predecessor objects, the function returns false if either the object in question is an OR-join and it satisfies the above two conditions (cf. lines 7 and 8 respectively), or it is not an OR-join and it is enabled as determined by function IsObjectEnabled (cf. line 10). If all the predecessors of the OR-join are visited and none satisfies any of these conditions, it means the OR-join should not wait for anything and thus the function returns true.

To analyze the algorithm's complexity, we first observe that function pred* involves computing a transitive closure which has a complexity of $O(|V| + |E|)$, E being the set of edges (flows in the BPD) and V the set of vertices (i.e. objects). Thus, the complexity of one invocation to this function is $O(N)$ where N is the total number of elements in the BPD. After computing the set of predecessors along each empty incoming flow of o, the algorithm iterates over this set of predecessors, which in the worst case includes all objects in the model except for end events. If one of these predecessors ($o′$) is itself an OR-join, the algorithm checks if there is a path between o and $o′$. This latter step involves invoking function pred* for each incoming flow of $o′$. We can thus bound the worst-case complexity of the algorithm by $O(N^2)$, as function pred* is potentially invoked for each element in the model, and each invocation has a cost of $O(N)$.

A substantial reduction in time complexity can be achieved by computing at design-time the set of predecessors of each object in the model, and storing the result in such a way that the set of predecessors of an object can be retrieved in constant time, e.g. using a hash table where the keys are objects in the BPD and the values are sets of predecessors. The size of this data structure is $O(T \times N)$, where T is the number of objects in the model (excluding flows). Once the data structure is materialized and the invocations to pred* are replaced by constant-time lookups, the complexity of the algorithm is reduced to $O(N)$.

5   Incremental Evaluation

Since the complexity of evaluating enablement for OR-joins is still higher than that for other gateways, it is desirable to further optimize the evaluation procedure. We therefore reuse the result of the evaluation of an OR-join's enablement in one state, when determining enablement of this OR-join in the next state. In other words, we would like to materialize the results of evaluating the enablement of each OR-join, so that after a state change (e.g. after an enabled object in the BPD fires), we only need to examine objects affected by the change. We call this *incremental evaluation*.

We assume a state change is represented as a set of flows in which tokens have been either added or removed. We capture all information pertaining to the enablement of each OR-join in a global (hash) table, namely *mustWaitFor*, which associates to each OR-join in the model, the set of predecessors for which this OR-join would have to wait if it was partially enabled, i.e. if it had at least one token in one of its incoming flows. For convenience, we write *mustWaitFor*[o] to refer to the entry in this table corresponding to object o, i.e. the set of predecessors that o must wait for as evaluated in the state prior to the change.

The incremental evaluation function is algorithmically described in Figure 7. The function takes as input an OR-join, the current state of the execution (after the change), and the state change Δ. The function should be called each time a state change occurs.

The first part of the algorithm (lines 2-11) updates the set *mustWaitFor*[o]. For each predecessor o' of o such that one of the incoming flows of o' has changed, the algorithm evaluates the new state of enablement of o' and, if necessary, it adds or removes o' from set *mustWaitFor*[o]. For this purpose, we reuse lines 7 and 8 of the *IsOrEnabled* algorithm, as well as function *IsObjectEnabled*, but we only call this latter function for objects other than OR-joins. In this "updating" phase of the algorithm, we consider predecessors of o along flows with no tokens as well as predecessors of o along flows that already contain tokens. The rationale is that all changes have to be accounted for, even if they do not have an immediate influence on the enablement of the OR-join o. Indeed, if o already has a token in one of its incoming flows, it will eventually fire, and when this happens, tokens will be removed from some of its incoming flows. As a result, some incoming flows may switch from having one token to having no token, and previously irrelevant changes may become relevant again.

1   **FUNCTION** IsOREnabledInc($o : \mathcal{G}^{\mathcal{I}}, tc : \mathcal{F} \to \mathbb{N}, \Delta : \wp(\mathcal{O} \times \mathcal{O})) : \mathbb{B}$
2   **foreach** $(o'', o') \in \Delta$ **where** $o' \in \text{pred*}(o) \setminus \{o\}$ **do**
3   **if** $o' \in \mathcal{G}^{\mathcal{I}} \wedge \exists po' \in pred(o') \; tc(po', o') \geq 1 \wedge$
4         $\forall w \in pred(o') \; o \in \text{pred*}(w) \Rightarrow tc(w, o') \geq 1$ **then**
5       $mustWaitFor[o] := mustWaitFor[o] \cup \{o'\}$;
6   **else if** $o' \notin \mathcal{G}^{\mathcal{I}} \wedge IsObjectEnabled(o', tc)$ **then**
7       $mustWaitFor[o] := mustWaitFor[o] \cup \{o'\}$;
8   **else**
9         $mustWaitFor[o] := mustWaitFor[o] \setminus \{o'\}$;
10   **end if**;
11   **end foreach**;
12   **if** $\exists po \in pred(o) \; tc(po, o) \geq 1$ **then**
13   **foreach** $po \in pred(o)$ **where** $tc(po, o) = 0$ **do**
14   **if** $\exists o' \in \text{pred*}(po) \; o' \in mustWaitFor[o]$ **then return** false;
15   **end foreach**
16   **return** true;
17   **else**
18   **return** false;
19   **end if**

Fig. 7. Algorithm for incrementally evaluating OR-join enablement.

Once all changes are accounted for, the status of the OR-join has to be re-evaluated (lines 12-19). Here, all predecessors along empty flows are checked. If at least one of them object is in the set $mustWaitFor[o]$, then o is not enabled.

6   Related Work

An assessment of fourteen state of the art commercial offerings in [15] revealed that only a handful of them support the OR-join construct without imposing syntactic restrictions. Many other languages support the OR-join but only in restricted settings. For example, in BPEL [9] it is only possible to define an OR-join in the context of acyclic networks of activities connected through so-called *control links*. In such restricted settings, the semantics of the OR-join becomes easier to define. Similarly, workflow management systems like InConcert, eProcess, and WebSphere MQ Workflow have avoided the problems related to defining the OR-join semantics by introducing syntactic restrictions. Eastman supports an OR-join with non-local semantics, but it is acknowledged in the Eastman manual that the use of OR-joins may result in poor performance [8].

  There are several proposals to formally define a semantics for the OR-join in Event Process Chains (EPCs) without introducing syntactic restrictions. In van der Aalst et al [1], the problems with the OR-join semantics in EPCs, especially that of vicious circles, are highlighted. It is suggested that any formal OR-join semantics will impose some restrictions or will deviate from the informal semantics to some extent. This leads to a proposal by Kindler [10,11] to define a non-local semantics in EPCs (including the OR-join) in terms of a pair of

transition relations using techniques from fixed point theory. Kindler's semantics can be considered as the most general and precise semantics of the OR-join, but as acknowledged by the author in subsequent papers, such fixed-point techniques are "very inefficient and intractable in practise" [5,6]

To address this computational complexity hurdle, Kindler proposed the use of binary decision diagrams (BDDs) to represent large sets of states and large transition relations in a compact manner [11]. Cuntz et al [4] later proposed a concrete method to calculate these transition systems using Kleene's fix-point theorem and techniques from symbolic model checking, instead of using a fixed-point iteration as in Kindler's original proposal. Specifically, Cuntz et al outline a technique for calculating the semantics of EPCs that combines a forward construction of the transition system with a backwards marking algorithm. Unfortunately, this optimized algorithm is not complete, meaning that it does not work on all EPC models. Also, despite these optimizations, the complexity of their approach is still proportional to the size of the transition system and hence, the approach only works for EPCs with small state spaces. In contrast, the OR-join semantics we have put forward can be evaluated in linear time on the number of elements in the process model, making it more scalable.

Mendling et al. [12] explore another approach to formally define the semantics of OR-joins in syntactically correct EPCs. This new semantics is inspired by the semantics of the OR-join in BPEL, but it can be applied to process models with arbitrary cycles. The semantics of Mendling et al. relies on the notion of state (i.e., tokens attached to arcs as in our approach) in combination with a notion of context (i.e., special tokens indicating if a given arc is in a "wait" or in a "dead" status). The context of an input arc to an OR-join is then used to determine whether an OR-join should be enabled at a given state. The evaluation of this OR-join semantics requires that both "normal" tokens, as well as "wait status" and "dead status" tokens are propagated during the execution of the process model. In contrast, our approach does not require the propagation of additional types of tokens. Also, in the semantics proposed by Mendling et al., an OR-join may "wait forever" if there is a deadlock preceding an OR-join. Specifically, if an AND-join connector "upstream" can not propagate tokens due to a deadlock, an OR-join connector "downstream" will continue to wait for a token forever. In contrast, our semantics will detect the deadlock situation upstream and the OR-join will be enabled rather than waiting for a token that will never arrive. In other words, the semantics of the OR-join presented in this paper detects deadlock situation and allows the OR-join to fire despite such deadlocks.

Wynn [19] proposed a general OR-join semantics for the YAWL workflow language that takes into account the "cancellation region" construct supported by this language. A concrete algorithm based on the backwards coverability of reset nets together with two optimization techniques are given to support the implementation in the YAWL workflow environment [18]. In line with Kindler's semantics, the OR-join semantics in YAWL is far-sighted: As soon as it is possible to conclude that an unmarked branch leading into an OR-join will not be reached, the OR-join will detect this situation and not wait anymore for tokens

along that branch. In contrast, the semantics proposed in this paper is more short-sighted: only when no state change whatsoever can occur among the set of predecessors leading into an incoming branch of an OR-join, will the OR-join fire. This choice constitutes a tradeoff between precision (i.e. how early do we detect that an OR-join can fire) and the efficiency of evaluating the enablement of an OR-join. Another difference is that the semantics proposed in this paper does not lead to deadlocks in the presence of vicious circles, while the YAWL semantics does. A major issue addressed by the OR-join semantics of YAWL is that of dealing with the notion of "cancellation feature". This is similar to the notion of "exception handler" in BPMN. However, in YAWL it is possible to cancel certain parts of a (sub)-process without canceling the entire (sub)-process. This leads to an interference between the OR-join behavior and the cancellation behavior. In BPMN, this type of cancellation behavior is impossible. Exception handling behavior in BPMN is attached to a sub-process and when an exception occurs, all the tokens from this sub-process are removed at once (and therefore all the OR-joins inside the sub-process are disabled). Therefore, the OR-join construct and the exception handling construct in BPMN do not interfere with one another.

In conclusion, our major contribution with respect to previous related work has been to define a semantics of the OR-join which has a linear computational complexity and can be evaluated incrementally, while at the same time not imposing syntactic restrictions on the process models and maintaining the desirable properties of a non-local OR-join semantics.

7   Conclusion

We have proposed a formalization of the enablement rules for a subset of BPMN objects, including the OR-join gateway, with the following characteristics:

– It does not impose restrictions on the topology of the process models, other than the minimal restrictions imposed by the syntax of BPMN itself.
– It has a relatively low computational complexity: determining if an OR-join is enabled can be computed in $O(N^2)$ where N is the total number of elements (objects + flows) in the model. This complexity can be reduced to $O(N)$ after materializing a data structure of size $O(N^2)$ at design-time. In addition, the semantics lends itself to an incremental evaluation mode.
– It does not generate deadlocks in the presence of cycles in the model involving multiple OR-joins.

The proposed OR-join semantics can be labeled as *operational* as it captures how an execution of a process model moves from one state to another. For static analysis purposes, it is desirable to have a semantics defined by translation of BPMN to a formalism (e.g. Petri nets) for which static analysis tools are available. In future, we plan to extend our mapping from BPMN to Petri nets [7] with the ability to deal with OR-joins. This would involve defining rules to transform BPMN models that contain OR-joins, into models that do not, since Petri nets

do not directly support this construct. As a by-product, such transformation rules could be useful in building simulation engines for BPMN, as they would replace all OR-joins with constructs that are easier to simulate.

References

1. van der Aalst, W.M.P., Desel, J., Kindler, E.: On the Semantics of EPCs: A Vicious Circle. In: Rump, M., Nüttgens, F.J. (eds.) Proceedings of the EPK 2002: Business Process Management using EPCs, Trier, Germany, pp. 71–80. Gesellschaft für Informatik (2002)
2. van der Aalst, W.M.P., ter Hofstede, A.H.M, Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distributed and Parallel Databases 14, 5–51 (2003)
3. van Breugel, F., Koshkina, M.: Models and verification of BPEL. Working paper (September 2006),
   `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf`
4. Cuntz, N., Freiheit, J., Kindler, E.: On the semantics of EPCs: Faster calculation for EPCs with small state spaces. In: Nüttgens, F.J., Rump, M. (eds.) Proceedings of EPK 2005, Hamburg, pp. 7–23 (December 2005)
5. Cuntz, N., Kindler, E.: On the semantics of EPCs: Efficient calculation and simulation. In: Nüttgens, F.J., Rump, M. (eds.) Proceedings of EPK 2004, pp. 7–26 (October 2004)
6. Cuntz, N., Kindler, E.: On the semantics of EPCs: Efficient calculation and simulation (Extended Abstract). In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 398–403. Springer, Heidelberg (2005)
7. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and automated analysis of BPMN process models. Preprint 5969, Queensland University of Technology (January 2007), `https://eprints.qut.edu.au/archive/00005969`
8. Eastman Software. RouteBuilder Tool User's Guide. Eastman Software, Inc, Billerica, MA, USA (1998)
9. Jordan, D., Evdemon, J. (eds.): Web Services Business Process Execution Language Version 2.0. OASIS WS-BPEL TC (2005),
   `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`
10. Kindler, E.: On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. In: Desel, J., Pernici, B., Weske, M. (eds.) BPM 2004. LNCS, vol. 3080, pp. 82–97. Springer, Heidelberg (2004)
11. Kindler, E.: On the Semantics of EPCs: Resolving the Vicious Circle. Data and Knowledge Engineering 56(1), 23–40 (2006)
12. Mendling, J., van der Aalst, W.M.P.: Formalization and Verification of EPCs with OR-Joins based on State and Context. In: CAiSE 2007. Proceedings of the 19th International Conference on Advanced Information Systems Engineering, Trondheim, Norway, Springer, Heidelberg (to appear, 2007)
13. OMG. Business Process Modeling Notation (BPMN) Version 1.0. OMG Final Adopted Specification. OMG (February 2006), `http://www.bpmn.org/`

14. Reis, S., Metzger, A., Pohl, K.: Integration testing in software product line engineering. In: FASE. Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering, Braga, Portugal, Springer, Heidelberg (2007)
15. Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., Mulyar, N.: Workflow control flow patterns: A revised view. BPMCenter Technical report BPM-06-22, BPMCenter.org (2006)
16. Silver, B.: The 2006 BPMS Report: Understanding and Evaluating BPM Suites (2006), `http://www.bpminstitute.org/bpmsreport.html`
17. Wong, P.Y.H., Gibbons, J.: A process semantics for BPMN. Preprint, Oxford University Computing Laboratory (March 2007),
    `http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/bpmn_extended.pdf`
18. Wynn, M.T.: Semantics, Verification, and Implementation of Workflows with Cancellation Regions and OR-joins. PhD Thesis, Faculty of Information Technology, Queensland University of Technology (November 2006)
19. Wynn, M.T., Edmond, D., van der Aalst, W.M.P., ter Hofstede, A.H.M: Achieving a General, Formal and Decidable Approach to the OR-join in Workflow using Reset nets. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 423–443. Springer, Heidelberg (2005)
20. Wynn, M.T., Edmond, D., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Achieving a General, Formal and Decidable Approach to the OR-join in Workflow using Reset nets. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 423–443. Springer, Heidelberg (2005)