Global vs. Local Semantics of BPMN 2.0 OR-Join

Flavio Corradini, Chiara Muzi(✉), Barbara Re, Lorenzo Rossi, and Francesco Tiezzi

School of Science and Technology, University of Camerino, Camerino, Italy
{flavio.corradini,chiara.muzi,barbara.re,lorenzo.rossi, francesco.tiezzi}@unicam.it

Abstract. Nowadays, BPMN 2.0 has acquired a clear predominance for modelling business processes. However, one of its drawback is the lack of a formal semantics, that leads to different interpretations, and hence implementations, of some of its features. This, as a matter of fact, results on process implementations using such features that do not fit with designers expectations, and that are not portable from one BPMN enactment tools to another. Among the BPMN elements particular ambiguous is the semantics of the OR-Join. Several formalisations of this element have been proposed in the literature, but none of them is derived from a direct and faithful translation of the current version of BPMN standard. In this work we instead provide direct, global and local, formalisations compliant with the OR-Join semantics reported in the BPMN 2.0 standard. In particular, the local semantics is devised to more efficiently determine the OR-Join enablement. The soundness of the approach is given by demonstrating the correspondence of the local semantics with respect to the global one.

1 Introduction

Nowadays, modelling is recognised as an important practice also in supporting software development. In particular, modelling business processes in complex organisations permits to better understand how organisations work and, at the same time, to support the development and continuous improvement of related IT systems [1]. In doing this, a challenge is to provide a precise semantics of the modelling languages used to guarantee that model behaviours do what they are supposed to do. We refer here to BPMN 2.0, the standard language for business process modelling [2]. Even if widely accepted, BPMN major drawbacks are related to the complexity of the BPMN meta-model semi-formal definition and to the possible misunderstanding of its execution semantics defined by means of natural text descriptions, sometimes containing misleading information [3]. These issues worsen when considering BPMN elements that have a particularly tricky behaviour, such as the OR-Join [4]. Roughly, this is used to synchronise two or more parallel flows according to specific (and non trivial) states on their execution status.

A M. Tjoa et al. (Eds.): SOFSEM 2018, LNCS 10706, pp. 321–336, 2018.
https://doi.org/10.1007/978-3-319-73117-9_23

This paper aims at formally specifying the OR-Join semantics of BPMN process models. This paves the way not only to formal reasoning, but also to driven implementations of process-aware IT systems ensuring an execution of the OR-Join compliant with BPMN 2.0. We focus on the OR-Join not only because of its semantic complexity, but also due to its practical impact, as that is a convenient way to relax the synchronisation of parallel control flows [5]. Its use is also confirmed by the number of models containing it (316 out of 7.541 BPMN 2.0 collaborations available in the BPM Academic Initiative public repository [6]).

In providing a novel formal semantics of the OR-Join specification we are firstly motivated by the results of our literature review on the topic (see Sect. 3). In fact, already available formalisation attempts mainly refer to previous versions of BPMN and do not fit with the current 2.0 standard (see [7–10]). Instead, those that rely on BPMN 2.0, such as [11], only consider the restricted class of sound processes. In addition, we have also practical motivations concerning the implementation of process-aware IT systems. We have experimented with some popular BPMN modelling and enactment tools and we have observed that most of them relax, simplify or even avoid the implementation of the OR-Join (see Sect. 3). In other words, almost all considered tools are not fully compliant with the OMG standard, thus resulting incompatible each other and not faithful with the designer expectations based on the BPMN specification.

Tackling the above issues, the contribution of this paper is twofold. Firstly, we provide a direct formalisation compliant with the OR-Join semantics reported in the current BPMN 2.0 standard specification. The semantics informally described in the specification is based on global information about the state of the whole process model. Thus, a direct, one-to-one, formalisation of this description has to be given with a *global* style, i.e., it is based on a notion of state storing information about tokens distribution over the whole model. From the practical point of view, however, this global perspective does not fit with the distributed nature of many process aware IT systems, where a single synchronisation point may not be aware of the execution state of the other process elements. Moreover, the naive implementation of the global conditions enabling the OR-Join would turn out to be quite inefficient. Thus, we also provide a *local* variant of the semantics, devised to more efficiently determine the OR-Join enablement, as it depends only on information local to the considered OR-Join. This semantics fosters a compositional, hence more scalable, approach for enacting processes with OR-Joins.

To sum up, the global semantics has been introduced as the formal reference, while the local one to be used for implementations. The soundness of our approach is given by the formal proof of their correspondence.

2 BPMN 2.0 Overview

Here we concentrate on those BPMN elements related to the process behaviour we use in the following. We also introduce a running example used throughout the paper.

BPMN Standard. BPMN process diagrams consist of combinations of different elements that can be organised in four classes (Fig. 1). **Events** are used to represent something that can happen; they can be used to start or end the process. **Gateways** are used to join (merging incoming sequence edges) or split (forking into outgoing sequence edges) the flow of a process. Three types of gateways are available XOR, AND and OR. An *XOR gateway* gives the possibility to describe choices; it is activated each time the gateway is reached and, when executed, it activates exactly one outgoing edge. An *AND gateway* has to wait to be reached by all its incoming edges to start, and then all the outgoing edges are started in parallel. A *OR gateway* has to wait to be reached by an arbitrary number of its incoming edges to start, and then at least one of the outgoing edges is started (see Sect. 3 for more details). **Tasks** are used to represent specific works to perform. Finally, **Sequence Edges** are used to specify the internal flow of the process, thus ordering elements.

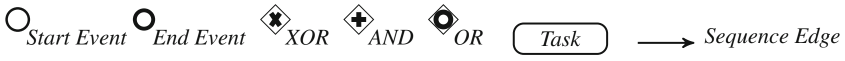

Fig. 1. Considered BPMN 2.0 elements.

A key concept related to the BPMN process execution is the notion of *token* [2, Sect. 7.1.1]. Commonly, a token traverses, from a start event, the sequence edges of the process and passes through its elements enabling their execution, and it is consumed by an end event when terminates. The distribution of tokens in the process elements is called *marking*, therefore the *process execution* is defined in terms of marking evolution.

Running Example. The elements illustrated above can be combined in order to design models like the one in Fig. 2 modelling an order fulfilment process. This is the case of a customer-oriented manufacturing caring about the quality of the order and accepting the payment only when the customer is fully satisfied. The process shown starts, due to the presence of a *start event*, whenever a purchase order has been received from a costumer. In order to manufacture a product, material availability is checked and then raw materials have to be ordered. Two preferred suppliers provide different types of raw materials. Depending on the product to be manufactured, raw materials may be ordered from either Supplier 1 or Supplier 2, or from both. This is rendered by including related *tasks* in a

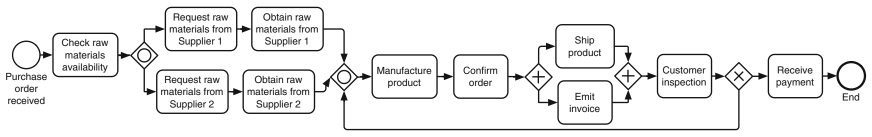
Purchase order received
Check raw materials availability
Request raw materials from Supplier 1
Obtain raw materials from Supplier 1
Request raw materials from Supplier 2
Obtain raw materials from Supplier 2
Manufacture product
Confirm order
Ship product
Emit invoice
Customer inspection
Receive payment
End

Fig. 2. An order fulfilment process diagram (revised version of model in [5]).

block composed of two OR *gateways*: an OR-Split, used to fork the flow into two branches after a decision; and an OR-Join that acts as a synchronisation point. Once raw materials are available, the product can be manufactured and the order confirmed. Then, tasks ‘Ship product’ and ‘Emit invoice’ can be performed independently from each other, so that they are put in a block between an AND-Split and an AND-Join enabling a parallel activation and a strict synchronisation before proceeding. The product is then inspected by the Customer: if he/she is unsatisfied, the product is manufactured again until he/she is pleased. Finally, when the Customer is satisfied the product is paid, and the process terminates by means of an *end event*.

In the rest of the paper and for the purpose of our study we intentionally left out tasks, since they do not affect the OR-Join execution [10]. Considering our running example, we get the process structure in Fig. 3.

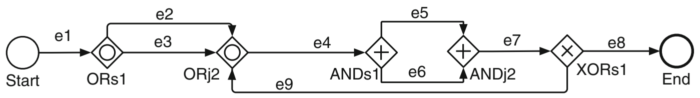

Fig. 3. The order fulfilment process structure.

3 Towards the OR-Join Formal Definition

Here we present in detail the semantics of BPMN 2.0 OR-Join as provided in the OMG specification. We also discuss related works, and give some preliminary notions we use throughout the paper to formalise the OR-Join behaviour.

From BPMN 2.0 Specification to Process Execution. The OR-Join semantics is quite complex, both from the definition point of view, in terms of formally expressing it, and from the computational point of view, in terms of determining whether an OR-Join is enabled. In our work we distil the characteristics of the OR-Join, from a detailed reading of the BPMN specification we report in Fig. 4 (where, as a matter of terminology, Inclusive Gateway stands for OR-Join, while Sequence Flow for sequence edge).

From the standard it is clear that the OR-Join has a *non-local semantics* and its activation may depend on the marking evolution considering the whole diagram. More in detail, given an OR-Join with a token in at least one of its incoming edges, it has to wait for a token that is in a path ending in a empty incoming edge of such OR-Join that does not visit the OR-Join itself. However, if this token is also in a path ending in a non-empty incoming edge, the OR-Join is activated and the execution can proceed.

The Inclusive Gateway is activated if:

- At least one incoming Sequence Flow has at least one token and
- For every directed path formed by sequence flow that:
  - (i) starts with a Sequence Flow *f* of the diagram that has a token,
  - (ii) ends with an incoming Sequence Flow of the inclusive gateway that has no token,
  - (iii) does not visit the Inclusive Gateway.
- There is also a directed path formed by Sequence Flow that:
  - (iv) starts with *f*,
  - (v) ends with an incoming Sequence Flow of the inclusive gateway that has a token,
  - (vi) does not visit the Inclusive Gateway.

Fig. 4. OR-Join semantics according to the OMG standard BPMN 2.0.

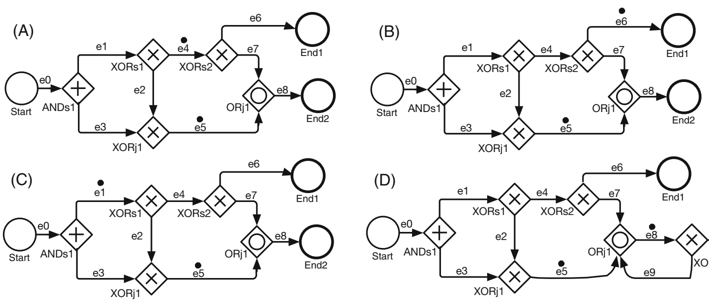

Fig. 5. OR-Join activation.

Let us consider the example in Fig. 5(A). In this case ORj1 has an incoming token in $\mathtt{e}_5$, but it is not activated because it has to wait for the token in $\mathtt{e}_4$ (corresponding to *f* in the definition in Fig. 4). Indeed, there is not another path from $\mathtt{e}_4$ to $\mathtt{e}_5$. However, if the token in $\mathtt{e}_4$ moves to $\mathtt{e}_6$, as in Fig. 5(B), the execution of ORj1 resumes, because now there is no marked path ending in $\mathtt{e}_7$. Moreover, if we move the token in $\mathtt{e}_4$ back to $\mathtt{e}_1$, as in Fig. 5(C), quite surprisingly ORj1 is activated, since this token can follow the path leading to $\mathtt{e}_5$. In this case, the OR-Join behaviour is quite anomalous; this is due to the fact that we are in presence of an unsafe model. Finally, to illustrate the effects of the condition "does not visit the *Inclusive Gateway*" in Fig. 4, let us consider a variant of the process where ORj1 is enclosed in a cycle (Fig. 5(D)). Also in this case ORj1 is activated; indeed, although the token in $\mathtt{e}_8$ is in a path ending in an empty edge incoming in ORj1 (i.e., $\mathtt{e}_9$), since it visits ORj1 this path is ignored.

OR-Join in the Literature. Most of the previous attempts to formalise the semantics of the OR-Join [7–10] are based on earlier versions of the BPMN standard, which provide different semantics for the OR-Join. Moreover, also when the same version of the standard is considered, different interpretations of the OR-Join behaviour, not always faithful to the specification, have been given. In particular, these differences regard the treatment of mutually dependent

OR-Joins (the so-called 'vicious circles') and of deadlock upstream an OR-Join. In fact, from a faithful translation of the standard, it results that mutually dependent OR-Joins are blocked, and that an OR-Join is not able to recognise that there is a deadlock on a path leading to it, thus it will wait forever. Below, we discuss the most significant related works.

Völzer [7] proposes a non-local semantics for the OR-Join in the BPMN 1.0 specification (2006) using workflow graphs. In case of vicious circles he argues that the intended meaning is not clear and hence they should be sort out by static analysis. This approach is then improved in [12], which quotes the 2010 version of the specification and gives an informal description of this one by means of inhibiting and anti-inhibiting paths. Dumas et al. [8] base their work on BPMN 1.0 and on the definition of the Synchronisation Merge pattern to which the specification refers to. They provide a local semantics, without imposing restrictions on the language, able to detect deadlocks upstream and to unlock mutually dependent OR-Joins. Thalheim et al. [9] make use of ASMs to introduce the OR-Join, by referring to the the specification of 2006, and make a comparison between the definitions given by other authors. Adopting a token-based view of workflow semantics, they start to analyse acyclic models. In this case, to threat the OR-Join, they introduce a special type of synchronisation tokens that fire flow objects in their downstream. They then consider cycles and, to deal with synchronisation in their presence, they introduce sets of tokens, which are viewed as a coherent group when a join fires. Christiansen et al. [10] refer to BPMN 2.0 - Beta 1, providing a global semantics directly in terms of a subset of BPMN. As for the vicious circle, they argue that, since informally BPMN specification does not include the resolution strategy and their work is a faithful translation, they do not consider it.

Differently from our work, the above approaches rely on past versions of the BPMN standard, which provide different semantics for the OR-Join with respect to the current 2.0 version. Thus, they cannot be applied as they are to the standard BPMN 2.0. Moreover, concerning the issues about vicious circles and deadlock upstream considered by some of those works, we have checked how they are dealt with by the current specification and, to be completely faithful with it, we have simply applied the same solution. Indeed, in the current description of the OR-Join semantics (Fig. 4), it does not seem to be any ambiguity about these two issues. The OR-Join is able to detect neither a vicious circle nor a deadlock upstream, thus in both cases its execution is blocked forever.

Recently, Prinz and Amme [11] propose a formalisation of the OR-Join semantics referred to the current version of the standard. However, they limit the work on *sound* workflow graphs, which identify a quite restricted class of BPMN processes [13]. In fact *soundness* is defined as the combination of properties concerning the dynamic behaviour of a process: option-to-complete, proper-completion, and no-dead-activities. Moreover, the proposed semantics does not fit with the standard as, for instance, it avoids vicious circles by determining which OR-Join in a circle has to wait and which one must proceed.

OR-Join Implementations. We have seen that in formalising the OR-Join semantics different interpretations have been given. The same has happened also for what concerns its implementation. Indeed, unfaithful implementations can be found in the most popular BPMN modelling and enactment tools. In particular, we have checked: Activiti [14], Camunda [15], Flowable [16], jBPM [17], ProcessMaker [18], Signavio [19], Stadust [20] and Sydle [21]. These BPMN tools provide their own interpretation of the BPMN standard, typically relaxing the OR-Join semantics. More specifically, Camunda and Flowable take advantage from the Activiti OR-Join implementation that in some cases keeps blocked a waiting token differently from what prescribed in the specification (see discussions above). A similar behaviour arises in Stadust. Instead, jBPM, Process Maker and Sydle relax the process structure handling only OR-Joins preceded by OR-Splits, and then enforce a simplified semantics. Last but not least, Signavio, and in particular its simulation feature, does not support the OR-Join at all.

Preliminaries. To define the formal semantics of a BPMN model we rely on information extracted from the model by means of a pre-processing step. This information consists of: *i.* paths from each OR-Join backward to the start event (and their suffix sub-paths) that do not visit the inclusive gateway; *ii.* sequence edges involved in a cycle; and *iii.* dependences between OR-Joins. We only consider models with one start event; this is not a limitation as in this setting each model can be rendered in this form.

For the purpose of our pre-processing, we consider a process model as a direct graph $G = (V, A)$ where: V is a set of *vertices*, ranged over by v and consisting of start events, end events, and gateways; and A is a set of *arrows*, consisting of triples (v_1, e, v_2) with $v_1 \neq v_2$ and $\mathsf{e} \in \mathbb{E}$, where $\mathbb{E}$ is the set of all (sequence) edges. Since edges are uniquely identified in a BPMN model, we have that for each (v_1, e, v_2) in A there exists no triple $(v_1', \mathsf{e}', v_2')$ in A with $\mathsf{e}' = \mathsf{e}$. This allows us to write, when convenient, (v_1, e, v_2) as e. Moreover, an OR-Join vertex is uniquely identified by the name of its outgoing edge.

A *path* in G, denoted by p, is a non-empty sequence of edges in A, where the third element of a triple is equal to the first of the next triple in the sequence, if any. A path that ends in its starting vertex is called *cycle*. For example, in the model in Fig. 3 we can observe the following cycle: $(\mathsf{e}_4, \mathsf{e}_6, \mathsf{e}_7, \mathsf{e}_9)$. Given a path p of the form $(v_0, \mathsf{e}_0, v_1), \ldots, (v_{k-1}, \mathsf{e}_{k-1}, v_k)$, notations $\mathsf{first}(p)$ and $\mathsf{last}(p)$ indicate the starting edge e_0 and the ending edge e_{k-1} of p, respectively.

We also refer with $\mathbb{P}$ the set of all the paths in G and we define $\mathcal{P} : \mathbb{E} \to 2^{\mathbb{P}}$ such a function that, given as input an edge $\mathsf{e} \in \mathbb{E}$ returns the set of all paths ending in the OR-Join uniquely identified by e and starting from all vertices between the start event and the OR-Join, which do not visit the considered OR-Join. Notably, this function returns a finite set of paths, because cycles within paths are not repeated. While computing $\mathcal{P}$, we can also compute the set $\mathbb{C} \subseteq \mathbb{E}$ of edges included in a cycle. Concerning the example in Fig. 3, we have $\mathcal{P}(\mathsf{e}_4) = \{(\mathsf{e}_2), (\mathsf{e}_3), (\mathsf{e}_1, \mathsf{e}_2), (\mathsf{e}_1, \mathsf{e}_3)\}$, and $\mathbb{C} = \{\mathsf{e}_4, \mathsf{e}_6, \mathsf{e}_7, \mathsf{e}_9\}$.

Finally, to properly formalise the OR-Join semantics in presence of vicious circles (i.e., to keep blocked the execution, see discussion above), we have to detect for each OR-Join the presence of OR-Joins from which it depends. This is expressed as a boolean predicate $\mathit{noDep} : \mathbb{E} \rightarrow \{\mathit{true}, \mathit{false}\}$, which taken as input an edge e identifying an OR-Join, it holds if no other OR-Join mutually depends with e.

To compute the pre-processing data mentioned above, we rely on existing graph theory procedures (the code is available at https://goo.gl/wv5Afu).

In particular, we use the *jGraphT* (www.jgrapht.org) Java library that is able to manage graphs. In this way, we capture cycles with the implementation of the Szwarcfiter and Lauer algorithm [22] and paths by using a Dijkstra-like algorithm [23].

4 Formalisation of the OR-Join Global Semantics

According to the OMG standard the semantics of the OR-Join requires global information about the state of the whole model. Here, we formalise this global perspective of the BPMN semantics. In particular, to enable a formal treatment of BPMN models including the OR-Join, we defined in Fig. 6 a BNF syntax of the model structure.

In the proposed grammar, the non-terminal symbol S represents *Process Structures*, while the terminal symbols, denoted by the $\mathsf{sans\ serif}$ font, are the considered elements of a BPMN model, i.e. events and gateways. The correspondence between the textual notation used here and the graphical notation of BPMN presented in Sect. 2 is as follows:

- $\mathsf{e} \in \mathbb{E}$ denotes a sequence edge, while $E \in 2^{\mathbb{E}}$ a set of edges; we require $|E| > 1$ when E is used in joining and splitting gateways;
- $\mathsf{start}(\mathsf{e})$ represents a start event with outgoing edge e;
- $\mathsf{end}(\mathsf{e})$ represents an end event with incoming edge e;
- $\mathsf{andSplit}(\mathsf{e}, E)$ (resp. $\mathsf{xorSplit}(\mathsf{e}, E)$, resp. $\mathsf{orSplit}(\mathsf{e}, E)$) represents an AND (resp. XOR, resp. OR) split gateway with incoming edge e and outgoing edges E;
- $\mathsf{andJoin}(\mathsf{e}, E)$ (resp. $\mathsf{xorJoin}(\mathsf{e}, E)$, resp. $\mathsf{orJoin}(\mathsf{e}, E)$) represents an AND (resp. XOR, resp. OR) join gateway with incoming edges E and outgoing edge e;
- $S_1 \mid S_2$ represents a composition of structure elements in order to render a process structure in terms of a collection of elements.

To achieve a compositional definition, each sequence edge of the BPMN model is split in two parts: the part outgoing from the source element and the part incoming into the target element. The two parts are correlated by means of unique sequence edge names in the BPMN model. To avoid malformed structure models, we only consider structures in which for each edge labelled by e outgoing from an element, there exists only one corresponding edge labelled by e incoming into another node, and vice versa.

$$\boxed{\begin{array}{l} S ::= \mathsf{start}(\mathsf{e}) \;\mid\; \mathsf{end}(\mathsf{e}) \;\mid\; \mathsf{andSplit}(\mathsf{e}, E) \;\mid\; \mathsf{andJoin}(\mathsf{e}, E) \;\mid\; \mathsf{xorSplit}(\mathsf{e}, E) \\ \quad \mid\; \mathsf{xorJoin}(\mathsf{e}, E) \;\mid\; \mathsf{orSplit}(\mathsf{e}, E) \;\mid\; \mathsf{orJoin}(\mathsf{e}, E) \;\mid\; S_1 | S_2 \end{array}}$$

Fig. 6. Syntax of BPMN process structures.

The operational semantics we propose is given in terms of configurations of the form $\langle S, \sigma, \mathcal{P}\rangle$, where: S is a process structure; σ is the execution state, storing for each edge the current number of tokens marking it; and $\mathcal{P}$ is the function that associates to each OR-Join gateway all paths that are incoming to it, not visiting it, and starting from marked edges (it results from pre-processing, see Sect. 3). Specifically, a state $\sigma : \mathbb{E} \rightarrow \mathbb{N}$ is a function mapping edges to numbers of tokens. The state obtained by updating in the state σ the number of tokens of the edge e to n, written as $\sigma \cdot \{\mathsf{e} \mapsto \mathsf{n}\}$, is defined by $(\sigma \cdot \{\mathsf{e}' \mapsto \mathsf{n}\})(\mathsf{e}) = \mathsf{n}$ if $\mathsf{e}' = \mathsf{e}$ and $\sigma(\mathsf{e})$ otherwise. The *inital state*, where all edges are unmarked, is denoted by σ_0; formally, $\sigma_0(\mathsf{e}) = 0 \;\; \forall \mathsf{e} \in \mathbb{E}$.

The reduction relation over configurations, written $\rightarrow_G$ and defined by the rules in Fig. 7, formalises the execution of a process in terms of edge marking evolution. Since such execution only affects the process state, for the sake of presentation, we omit the structure and $\mathcal{P}$ from the target configuration of the transition. Moreover, since $\mathcal{P}$ is exploited only by the OR-Join rule, it will also be omitted from the source configuration. Thus, $\langle S, \sigma, \mathcal{P}\rangle \rightarrow_G \langle S, \sigma', \mathcal{P}\rangle$ shall be usually written as $\langle S, \sigma\rangle \rightarrow_G \sigma'$. Before commenting on the rules, we introduce the auxiliary functions they exploit. Function $inc : \mathbb{S} \times \mathbb{E} \rightarrow \mathbb{S}$ (resp. $dec : \mathbb{S} \times \mathbb{E} \rightarrow \mathbb{S}$), where $\mathbb{S}$ is the set of states, allows updating a state by incrementing (resp. decrementing) by one the value of an edge in the state. Formally, they are defined as follows: $inc(\sigma, \mathsf{e}) = \sigma \cdot \{\mathsf{e} \mapsto \sigma(\mathsf{e}) + 1\}$ and $dec(\sigma, \mathsf{e}) = (\sigma \cdot \{\mathsf{e} \mapsto \sigma(\mathsf{e}) - 1\}$. These functions extend to sets of edges as follows: $inc(\sigma, \varnothing) = \sigma$ and $inc(\sigma, \{\mathsf{e}\} \cup E)) = inc(inc(\sigma, \mathsf{e}), E)$ (the cases for dec are similar).

We now briefly comment on the operational rules. Rule *G-Start* starts the execution of a process when it is in its initial state (i.e., all edges are unmarked). The effect of the rule is to increment the number of tokens in the edge outgoing from the start event. For the sake of simplicity, the rule is defined in a way that, when the process execution terminates it can restart. Rule *G-End* instead is enabled when there is at least a token in the incoming edge of the end event, which is then removed. Rule *G-AndSplit* is applied when there is at least one token in the incoming edge of an AND-Split gateway; as result of its application the rule decrements the number of tokens in the incoming edge and increments that in each outgoing edge. Similarly, rule *G-AndJoin* decrements the tokens in each incoming edge and increments the number of tokens of the outgoing edge, when each incoming edge has at least one token. Rule *G-XorSplit* is applied when a token is available in the incoming edge of a XOR-Split gateway, the rule decrements the token in the incoming edge and increment the token in one of the outgoing edges. Rule *G-XorJoin* is activated every time there is a token in one of the incoming edges, which is then moved to the outgoing edge. Rule *G-OrSplit*

$\langle \mathsf{start}(\mathsf{e}), \sigma_0 \rangle \rightarrow_G inc(\sigma_0, \mathsf{e})$		(*G-Start*)
$\langle \mathsf{end}(\mathsf{e}), \sigma \rangle \rightarrow_G dec(\sigma, \mathsf{e})$	$\sigma(\mathsf{e}) > 0$	(*G-End*)
$\langle \mathsf{andSplit}(\mathsf{e}, E), \sigma \rangle \rightarrow_G inc(dec(\sigma, \mathsf{e}), E)$	$\sigma(\mathsf{e}) > 0$	(*G-AndSplit*)
$\langle \mathsf{andJoin}(\mathsf{e}, E), \sigma \rangle \rightarrow_G inc(dec(\sigma, E), \mathsf{e})$	$\forall \mathsf{e}' \in E \,.\, \sigma(\mathsf{e}') > 0$	(*G-AndJoin*)
$\langle \mathsf{xorSplit}(\mathsf{e}, \{\mathsf{e}'\} \cup E), \sigma \rangle \rightarrow_G inc(dec(\sigma, \mathsf{e}), \mathsf{e}')$	$\sigma(\mathsf{e}) > 0$	(*G-XorSplit*)
$\langle \mathsf{xorJoin}(\mathsf{e}, \{\mathsf{e}'\} \cup E), \sigma \rangle \rightarrow_G inc(dec(\sigma, \mathsf{e}'), \mathsf{e})$	$\sigma(\mathsf{e}') > 0$	(*G-XorJoin*)
$\langle \mathsf{orSplit}(\mathsf{e}, E_1 \sqcup E_2), \sigma \rangle \rightarrow_G inc(dec(\sigma, \mathsf{e}), E_1)$	$\sigma(\mathsf{e}) > 0 \quad E_1 \neq \varnothing$	(*G-OrSplit*)
$\langle \mathsf{orJoin}(\mathsf{e}, E_1 \sqcup E_2), \sigma \rangle \rightarrow_G inc(dec(\sigma, E_1), \mathsf{e})$	$\forall \mathsf{e}' \in E_1 \,.\, \sigma(\mathsf{e}') > 0$ $\forall \mathsf{e}' \in E_2 \,.\, \sigma(\mathsf{e}') = 0$ $E_1 \neq \varnothing \qquad \forall p_1 \in \Pi \,.\, \exists\, p_2 \in \Pi_{p_1}$	(*G-OrJoin*)

$$\frac{\langle S_1, \sigma \rangle \rightarrow_G \sigma'}{\langle S_1 \mid S_2, \sigma \rangle \rightarrow_G \sigma'}\ (\textit{G-Int}_1) \qquad \frac{\langle S_2, \sigma \rangle \rightarrow_G \sigma'}{\langle S_1 \mid S_2, \sigma \rangle \rightarrow_G \sigma'}\ (\textit{G-Int}_2)$$

Fig. 7. BPMN global semantics.

is activated when there is a token in the incoming edge of an OR-Split gateway, which is then removed while a token is added in some outgoing edges (at least one). Notably, in the rule we make use of operator $\sqcup$, denoting the disjoint union of sets, i.e. $E_1 \sqcup E_2$ stands for $E_1 \cup E_2$ if $E_1 \cap E_2 = \varnothing$, it is undefined otherwise. Rules *G-Int₁* and *G-Int₂* deal with interleaving in a standard way.

We conclude by describing in detail the rule *G-OrJoin* defining the semantics of the OR-Join gateway. The operator $\sqcup$ is used to split the set of edges incoming in the OR-Join into two disjoint sets, E_1 and E_2, such that one contains marked edges ($\forall \mathsf{e}' \in E_1 \,.\, \sigma(\mathsf{e}') > 0$) and the other one contains unmarked edges ($\forall \mathsf{e}' \in E_2 \,.\, \sigma(\mathsf{e}') = 0$). In describing the rule we quote the BPMN 2.0 specification to make clear the correspondence. *"The Inclusive Gateway is activated if"* the conditions for the rule applications are satisfied. Thus, the requirement *"At least one incoming Sequence Flow has at least one token"* is represented by condition $E_1 \neq \varnothing$. The second requirement *"For every directed path formed by Sequence Flow that (i)... (ii)... (iii)... There is also a directed path formed by Sequence Flow that (iv)... (v)... (vi)"* is represented by the condition $\forall p_1 \in \Pi \,.\, \exists\, p_2 \in \Pi_{p_1}$, where Π is the set of paths satisfying (*i*), (*ii*) and (*iii*), while the sets Π_p, one for each path p in Π, contain paths satisfying (*iv*), (*v*) and (*vi*). Formally, they are defined as $\Pi = \{p \in \mathcal{P}(\mathsf{e}) \mid \sigma(\mathsf{first}(p)) > 0 \wedge \mathsf{last}(p) \in E_2\}$ and $\Pi_p = \{p' \in \mathcal{P}(\mathsf{e}) \mid \mathsf{first}(p') = \mathsf{first}(p) \wedge \mathsf{last}(p') \in E_1\}$. In particular, a path p in Π is such that: "*(i) starts with a Sequence Flow f of the diagram that has a token*" ($\sigma(\mathsf{first}(p)) > 0$), "*(ii) ends with an incoming Sequence Flow of the inclusive gateway that has no token*" ($\mathsf{last}(p) \in E_2$), and "*(iii) does not visit the Inclusive Gateway*" (ensured by definition of $\mathcal{P}$). Instead, given a path p in Π, a path p' in Π_p is such that: "*(iv) starts with f*" ($\mathsf{first}(p') = \mathsf{first}(p)$, as f is the first edge of p), "*(v) ends with an incoming Sequence Flow of the inclusive gateway that has a token*" ($\mathsf{last}(p') \in E_1$), and "*(vi) does not visit the Inclusive Gateway*" (ensured again by definition of $\mathcal{P}$).

Example 1. The initial configuration of the process in Fig. 3 is $\langle S, \sigma_0 \rangle$ where:

$$S = \mathsf{start}(\mathsf{e}_1) \mid \mathsf{orSplit}(\mathsf{e}_1, \{\mathsf{e}_2, \mathsf{e}_3\}) \mid \mathsf{orJoin}(\mathsf{e}_4, \{\mathsf{e}_2, \mathsf{e}_3, \mathsf{e}_9\}) \mid \mathsf{andSplit}(\mathsf{e}_4, \{\mathsf{e}_5, \mathsf{e}_6\}) \\ \mid \mathsf{andJoin}(\mathsf{e}_7, \{\mathsf{e}_5, \mathsf{e}_6\}) \mid \mathsf{xorSplit}(\mathsf{e}_7, \{\mathsf{e}_8, \mathsf{e}_9\}) \mid \mathsf{end}(\mathsf{e}_8)$$

By applying rule*G-Start* the execution of the process starts by marking with a token the edge e_1. Rule *G-OrSplit* can be then applied; it moves the token from e_1 to one (or more) outgoing edges of the OR-Split, say e_3. Now, all premises of rule *G-OrJoin* are satisfied: $E_1 = \{\mathsf{e}_3\} \neq \varnothing$, and the condition based in the universal quantification trivially holds as $\Pi = \varnothing$, since all paths with a token at the beginning and no token at the end, e.g. $(\mathsf{e}_3, \mathsf{e}_4, \mathsf{e}_5, \mathsf{e}_7, \mathsf{e}_9)$, do visit the OR-Join, thus violating the requirement (*iii*). Therefore, the rule can be applied and the token in e_3 moves to e_4. From there, the execution simply proceeds according to the semantics of AND and XOR gateways.

5 Formalisation of the OR-Join Local Semantics

The OR-Join semantics presented in the previous section perfectly fits with the informal definition given in the BPMN 2.0 standard specification. However, the evaluation of the OR-Join gateway activation (formalised by the premises of rule *G-OrJoin*) requires a global view of the process marking. From a practical perspective, this may complicate the implementation of the process control flow, also considering that the semantics of all other BPMN constructs is *local*, i.e. it relies only on the information about the marking of incoming and outgoing edges. Therefore, we propose in this section an alternative, yet equivalent, semantics of BPMN, including the OR-Join construct, that is local.

For the local semantics, we consider only *safe* models [24]. Safeness requires a model to not activate an edge more than once at the same time. This assumption is not too restrictive, since safeness is recognized as one of the most important correctness criteria for business process models [25]. The lack of this property, in fact, may cause issues concerning process execution, related e.g. to the proper termination of processes or to erroneous synchronizations among concurrent control flows [26].

To enable local treatment of the BPMN semantics, roughly the global state information of a process is spread over the edges of its structure, resulting on a *Marked Process*. Formally, the syntax of marked processes, denoted by M, is defined in Fig. 8. The only difference between the syntax of a marked process and a process structure is that in the former an edge is also characterised by a *type* T, indicating if it is part of a cycle (c) or not (nc), and by a *status* Σ, denoting whether a token is marking the edge (*live* status denoted by l), or may still arrive (*wait* status denoted by w), or will not arrive (*dead* status denoted by d). As

$$\begin{array}{l}
M ::= \mathsf{start}(\mathsf{e}.T.\Sigma) \mid \mathsf{andSplit}(\mathsf{e}.T.\Sigma, E) \mid \mathsf{andJoin}(\mathsf{e}.T.\Sigma, E) \mid \mathsf{xorSplit}(\mathsf{e}.T.\Sigma, E) \\
\qquad \mid \mathsf{xorJoin}(\mathsf{e}.T.\Sigma, E) \mid \mathsf{orSplit}(\mathsf{e}.T.\Sigma, E) \mid \mathsf{orJoin}(\mathsf{e}.T.\Sigma, E) \mid \mathsf{end}(\mathsf{e}.T.\Sigma) \mid M_1 \mid M_2 \\
T ::= \mathsf{c} \quad \mid \quad \mathsf{nc} \qquad\qquad \Sigma ::= \mathsf{l} \quad \mid \quad \mathsf{w} \quad \mid \quad \mathsf{d}
\end{array}$$

Fig. 8. BPMN syntax of marked processes.

explained in Sect. 3, edge types are statically determined in the pre-processing. With abuse of notation, edge set notation E extends to marked edges.

Now, the operational semantics does not need to consider any more configurations with a state, but it is directly given in terms of marked processes. Formally, the operational semantics is defined by means of a labelled transition relation $M \xrightarrow{\alpha}_L M'$, meaning that "the marked process M performs a transition labelled by α and becomes M' in doing so". Labels α are used to propagate the effect of marking updates, resulting from the evolution of a subterm of the process, to the other subterms. They are triples of the form $(\mathsf{w} : E_1, \mathsf{d} : E_2, \mathsf{l} : E_3)$, indicating the edges whose status must be set to w, d and l, respectively. For the sake of simplicity, within labels, sets E_i contain just edge names (without type and status). Moreover, to improve readability, we omit a field of the triple when the associated edge set is empty, and we remove brackets $\{$and$\}$ in case of singleton; for example, the label $(\mathsf{w} : \varnothing, \mathsf{d} : \varnothing, \mathsf{l} : \{\mathsf{e}\})$ is written $\mathsf{l} : \mathsf{e}$. Finally, to identify the initial status of a marked process M we rely on the boolean predicate $\mathit{isInit}(M)$, which holds when all edges of M have status w. Due to lack of space, we present below an excerpt of the operational semantics; we refer the interested reader to the companion technical report [27] for a complete account of definitions, operational rules, proofs of the correspondence results, and application to the running examples.

To define the labelled transition relation, we need a few auxiliary functions. First, we exploit $\mathit{setDead}(E)$ and $\mathit{setWait}(E)$ to change the status of gateway edges to d and w, respectively. Similarly, to check if the edges in E have live (resp. dead) status, we make use of the boolean function $\mathit{isLive}(E)$ (resp. $\mathit{isDead}(E)$). Finally, to distinguish the type T of edges in E we make use of boolean functions $\mathit{isC}(E)$ (resp. $\mathit{isNC}(E)$). All these functions are inductively defined on the structure of E.

In Fig. 9 we report some significant operational rules defining the evolution of live tokens in the BPMN local semantics. A start event has edges with non-cyclic type, as according to the BPMN standard it cannot have an incoming edge. Rule *L-Start-NC* annotates the edge e outgoing from the start event with l when the process is in the initial status (in fact, the edge has a w status before the transition), the corresponding label $\mathsf{l} : \mathsf{e}$ is produced. Let us consider the OR-Join rules. *L-OrJoin-NC* is applied when the outgoing edge is of non-cyclic type, while L_1-*OrJoin-C* and L_2-*OrJoin-C* when it is of type c. In these latter cases, we also make use of the boolean predicate $\mathit{noDep}(\mathsf{e})$, defined in Sect. 3, to ensure that in case of vicious circles ($\mathit{noDep}(\mathsf{e}) = \mathit{false}$) the rules cannot be applied, thus enforcing a deadlocked behaviour as prescribed by BPMN 2.0.

$$\mathsf{start}(\mathsf{e.nc.w}) \xrightarrow{\mathsf{l{:}e}}_L \mathsf{start}(\mathsf{e.nc.l}) \qquad (\textit{L-Start-NC})$$

$$\begin{array}{l} \mathsf{orJoin}(\mathsf{e.nc.}\Sigma, E_1 \sqcup E_2) \xrightarrow{(\mathsf{d}:E_1 \sqcup E_2, \mathsf{l{:}e})}_L \\ \quad \mathsf{orJoin}(\mathsf{e.nc.l}, \mathit{setDead}(E_1 \sqcup E_2)) \end{array} \quad \begin{array}{l} E_1 \neq \emptyset \ \mathit{isLive}(E_1) \\ \mathit{isDead}(E_2) \end{array} \quad (\textit{L-OrJoin-NC})$$

$$\begin{array}{l} \mathsf{orJoin}(\mathsf{e.c.}\Sigma, E_1 \sqcup E_2) \xrightarrow{(\mathsf{w}:E_1, \mathsf{d}:E_2, \mathsf{l{:}e})}_L \\ \quad \mathsf{orJoin}(\mathsf{e.c.l}, \mathit{setWait}(E_1) \sqcup \mathit{setDead}(E_2)) \end{array} \quad \begin{array}{l} E_1 \neq \emptyset \ \mathit{isLive}(E_1) \\ \mathit{isDead}(E_2) \ \mathit{isC}(E_1) \\ \mathit{isNC}(E_2) \ \mathit{noDep}(\mathsf{e}) \end{array} \quad (\textit{L}_1\textit{-OrJoin-C})$$

$$\begin{array}{l} \mathsf{orJoin}(\mathsf{e.c.}\Sigma, E_1 \sqcup E_2) \xrightarrow{(\mathsf{w}:E_2, \mathsf{d}:E_1, \mathsf{l{:}e})}_L \\ \quad \mathsf{orJoin}(\mathsf{e.c.l}, \mathit{setDead}(E_1) \sqcup \mathit{setWait}(E_2)) \end{array} \quad \begin{array}{l} E_1 \neq \emptyset \ \mathit{isLive}(E_1) \\ \mathit{isDead}(E_2) \ \mathit{isNC}(E_1) \\ \mathit{isC}(E_2) \ \mathit{noDep}(\mathsf{e}) \end{array} \quad (\textit{L}_2\textit{-OrJoin-C})$$

$$\mathsf{orJoin}(\mathsf{e}.T.\Sigma, E) \xrightarrow{\mathsf{d{:}e}}_L \mathsf{orJoin}(\mathsf{e}.T.\mathsf{d}, E) \qquad \mathit{isDead}(E) \qquad (\textit{D-OrJoin})$$

$$\frac{M_1 \xrightarrow{\alpha}_L M_1'}{M_1 \mid M_2 \xrightarrow{\alpha}_L M_1' \mid M_2 \star \alpha} \quad (\textit{M-StatusUpd}_1)$$

Fig. 9. BPMN local semantics (an excerpt).

The rules described so far are not enough for properly expressing the OR-Join behaviour only using local information. Other rules are indeed needed to propagate the dead status. They are applied when all incoming edges of a gateway are annotated with d, and propagate this information to the outgoing edges. As an example, we report here rule *D-OrJoin*. Finally, *M-StatusUpd* allows the interleaving of the process element. It relies on the status updating function $M \star \alpha$, which returns a process obtained from M by updating the status of its edges according to the labelled sets they belong to in α.

We conclude the section with our main result, ensuring the soundness of our approach. In particular, we show the correspondence between the global and local semantics we provided. In order to do that we first need to illustrate the correspondence between the syntax used in the global formalisation and that used in the local version. The local notation is achieved by applying σ to the structure S, that is by distributing the token information included in σ on the edges of S. We recall, we consider only safe processes, thus $0 \leqslant \sigma(\mathsf{e}) \leqslant 1$. Formally, we have the following definition; we rely here on auxiliary notations t and nl to denote an *undefined type*, which can be either c or nc, and a *not live* status, which can be either w or d.

Definition 1 (Syntax correspondence). *Let $\langle S, \sigma\rangle$ be a process configuration, then $S \cdot \sigma$ is inductively defined on the structure of S as follows (we show here only few cases of the definition, since the other are similar):*

$$\mathsf{start}(\mathsf{e}) \cdot \sigma = \mathsf{start}(\mathsf{e.t.}(\mathsf{e} \cdot \sigma)) \qquad \mathsf{end}(\mathsf{e}) \cdot \sigma = \mathsf{end}(\mathsf{e.t.}(\mathsf{e} \cdot \sigma))$$

$$\mathsf{orJoin}(\mathsf{e}, E) \cdot \sigma = \mathsf{orJoin}(\mathsf{e.t.}(\mathsf{e} \cdot \sigma), (E \cdot \sigma)) \qquad (S_1 \mid S_2) \cdot \sigma = S_1 \cdot \sigma \mid S_2 \cdot \sigma$$

$$\textit{where } \mathsf{e} \cdot \sigma = \begin{cases} \mathsf{l} \text{ if } \sigma(\mathsf{e}) = 1; \\ \mathsf{nl} \text{ otherwise.} \end{cases} \qquad \emptyset \cdot \sigma = \emptyset \qquad (\{\mathsf{e}\} \cup E) \cdot \sigma = \{\mathsf{e} \cdot \sigma\} \cup (E \cdot \sigma).$$

According to the above definition, a term $S \cdot \sigma$ represents a class of marked processes, i.e. all those processes with the same marking for what concerns the live status, but possibly different markings for the other two status and possibly different edge types (information that indeed are not considered at all in the global semantics). Therefore, to state that marked processes belong to a given class we use the relation $\equiv$, whose meaning is as follows: $M \equiv S \cdot \sigma$ means that M is syntactical equivalent to $S \cdot \sigma$, up to an instantiation of t and nl occurrences in $S \cdot \sigma$.

Finally, our results rely on the notion of *reachable* configuration/processes. In fact, the considered syntaxes are too liberal, as they allow terms that cannot be obtained (by means of transitions) from a process in its initial state.

Definition 2 (Reachable configuration/marked process). *A process configuration $\langle S, \sigma\rangle$ (resp. marked process M) is* reachable *if there exists $\langle S, \sigma'\rangle$ (resp. process M') such that $\sigma' = \sigma_0$ (resp. isInit(M')) and $\langle S, \sigma'\rangle \rightarrow_G^* \sigma$ (resp. $M' \xrightarrow{\alpha}_L^* M$).*

Now, we can formally define our results, stating that each step of the global semantics corresponds to one or more steps of the local semantics (Theorem 1) and vice versa (Theorem 2). Their proofs are given by induction on the derivation of the transitions.

Theorem 1. *Let $\langle S, \sigma\rangle$ be a reachable process configuration, if $\langle S, \sigma\rangle \rightarrow_G \sigma'$ then there exists M such that $M \equiv S \cdot \sigma$, $M \xrightarrow{\alpha}_L^+ M'$ and $M' \equiv S \cdot \sigma'$.*

Theorem 2. *Let M be a reachable marked process, with $M \equiv S \cdot \sigma$, if $M \xrightarrow{\alpha}_L M'$, then there exists M'' such that $M' \xrightarrow{\alpha}_L^* M''$, $\langle S, \sigma\rangle \rightarrow_G \sigma'$ and $M'' \equiv S \cdot \sigma'$.*

6 Concluding Remarks

In this paper we presented global and local direct formalisations of BPMN process models compliant with the OR-Join semantics reported in the BPMN 2.0 standard. In particular, the local semantics fosters a compositional, and hence more scalable, approach to enact business processes involving OR-Joins. The soundness of the proposed approach is given by the formal correspondence between the local and global semantics.

As a future work, we plan to validate the performance of the proposed global and local semantics over models coming from real scenarios. Moreover, we intend to use the OR-Join semantics to enable process verification and ensure process models correctness by design. Last but not least, we plan to extend enactment tools, such as Camunda [15], to implement process aware IT systems fitting with the proposed semantics.

References

1. Pastor, O.: Model-driven development in practice: from requirements to code. In: Steffen, B., Baier, C., van den Brand, M., Eder, J., Hinchey, M., Margaria, T. (eds.) SOFSEM 2017. LNCS, vol. 10139, pp. 405–410. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51963-0_31
2. OMG: Business Process Model and Notation (BPMN V 2.0) (2011)
3. Suchenia, A., Potempa, T., Ligęza, A., Jobczyk, K., Kluza, K.: Selected approaches towards taxonomy of business process anomalies. In: Pełech-Pilichowski, T., Mach-Król, M., Olszak, C.M. (eds.) Advances in Business ICT: New Ideas from Ongoing Research. SCI, vol. 658, pp. 65–85. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-47208-9_5
4. van der Aalst, W.M., Desel, J., Kindler, E.: On the semantics of EPCs: a vicious circle. In: EPK, pp. 71–79 (2002)
5. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-33143-5
6. Kunze, M., Berger, P., Weske, M.: BPM academic initiative - fostering empirical research. In: BPM Demonstration Track, CEUR Workshop Proceedings, vol. 940, pp. 1–5 (2012)
7. Völzer, H.: A new semantics for the inclusive converging gateway in safe processes. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 294–309. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15618-2_21
8. Dumas, M., Grosskopf, A., Hettel, T., Wynn, M.: Semantics of standard process models with OR-Joins. In: Meersman, R., Tari, Z. (eds.) OTM 2007. LNCS, vol. 4803, pp. 41–58. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76848-7_5
9. Thalheim, B., Sorensen, O., Borger, E.: On defining the behavior of OR-Joins in business process models. J. Univ. Comput. Sci. **15**(1), 3–32 (2009)
10. Christiansen, D.R., Carbone, M., Hildebrandt, T.: Formal semantics and implementation of BPMN 2.0 inclusive gateways. In: Bravetti, M., Bultan, T. (eds.) WS-FM 2010. LNCS, vol. 6551, pp. 146–160. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19589-1_10
11. Prinz, T.M., Amme, W.: A complete and the most liberal semantics for converging OR gateways in sound processes. Complex Syst. Inf. Model. Q. **4**, 32–49 (2015). http://dblp.org/db/journals/csimq/csimq4
12. Gfeller, B., Völzer, H., Wilmsmann, G.: Faster Or-Join enactment for BPMN 2.0. In: Dijkman, R., Hofstetter, J., Koehler, J. (eds.) BPMN 2011. LNBIP, vol. 95, pp. 31–43. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25160-3_3
13. Wynn, M.T., et al.: Business process verification-finally a reality!. Bus. Process Manag. J. **15**(1), 74–92 (2009)
14. Alfresco Software Inc: Activiti v. 6.0. www.activiti.org (2017)
15. Camunda services GmbH: Camunda v. 7.7.0. www.camunda.com (2017)
16. Flowable: Flowable v. 6.1.0. www.flowable.org (2017)
17. Red Hat: jBPM v. 7.0.0. www.jBPM.org (2017)
18. ProcessMaker Inc.: Process maker v. 3.2. www.processmaker.com (2017)
19. Signavio Inc: Signavio v. 11.2.0. www.signavio.com (2017)
20. Stadust: Stadust v. 4.1.0. www.eclipse.org/stardust (2017)
21. Sydle: Sydle. www.sydle.com (2017)

22. Szwarcfiter, J.L., Lauer, P.E.: A search strategy for the elementary cycles of a directed graph. BIT Numer. Math. **16**(2), 192–204 (1976)
23. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
24. van der Aalst, W.M.P.: Workflow verification: finding control-flow errors using petri-net-based techniques. In: van der Aalst, W., Desel, J., Oberweis, A. (eds.) Business Process Management. LNCS, vol. 1806, pp. 161–183. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45594-9_11
25. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. **50**(12), 1281–1294 (2008)
26. Corradini, F., Fornari, F., Muzi, C., Polini, A., Re, B., Tiezzi, F.: On avoiding erroneous synchronization in BPMN processes. In: Abramowicz, W. (ed.) BIS 2017. LNBIP, vol. 288, pp. 106–119. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59336-4_8
27. Corradini, F., Muzi, C., Re, B., Rossi, L., Tiezzi, F.: Global vs. Local Semantics of BPMN 2.0 OR-Join. Technical report, Univ. Camerino (2017). http://pros.unicam.it/documents.html