Complex Systems Informatics and Modeling Quarterly
CSIMQ, Issue 4, October, 2015, Pages 32-49
Published online by RTU Press, https://csimq-journals.rtu.lv
http://dx.doi.org/10.7250/csimq.2015-4.03
ISSN: 2255-9922 online

A Complete and the Most Liberal Semantics for Converging OR Gateways in Sound Processes

Thomas M. Prinz and Wolfram Amme

Friedrich Schiller University Jena, 07743 Jena, Germany

Thomas.Prinz@uni-jena.de, Wolfram.Amme@uni-jena.de

Abstract. Although the semantics of converging OR gateways (also known as OR-joins) in business processes is far from trivial, they are frequently used. In this paper, we describe a general definition for soundness of processes guaranteeing the absence of deadlocks and no lack of synchronization for each possible OR-join semantics. Then, we derive a criterion — completeness — to evaluate existing approaches of OR-join semantics. As a result, no currently existing OR-join semantics is complete; therefore, there are actually correct processes being not successfully executable. For that, we provide our own approach based on a traditional relation of compiler construction; and we show that this approach is complete and can be called the most liberal possible.

Keywords: Business process, OR-join, completeness, semantics, soundness.

1 Introduction

Graphical notations of programs and processes allowing explicit parallelism, e.g., Business Process Model and Notation (BPMN) [1] or Event-driven Process Chains, are standard during the modeling of (business) processes in software engineering and business process management. Such a modeling of processes is a difficult and error-prone task. Especially, the fusion of different paths is sometimes complex, so that often it is the best choice to use an inclusive converging (OR) gateway — for reasons of correctness and readability, e.g., because of compactness. However, the semantics of the converging OR gateways, called OR-joins, is one of the major problems in defining an automatic and correct execution of arbitrary processes [2].

The major task of OR-joins in practice is the synchronization of an arbitrary non-empty set of control flows. Figure 1 illustrates two processes in BPMN notation containing OR-joins. The left-hand process *(a)* has an OR diverging gateway OS_1 (an OR-split) making it possible to execute an individual non-empty subset of its successor tasks T_1, T_2, and T_3. The OR-join OJ_1 then combines these flows into a single one. In this case, this OR-join only has to wait for the arrival of all incoming control flows, i.e., an OR-join has a non-local semantics since its execution does not only depend on the control of its incoming edges. Moreover, an OR-join has to wait for all control flows which may still arrive at its incoming edges. This informal description of *'may still arrive'* implies possible future states. However, they can only be determined if the semantics of OR-joins exists. Therefore, a semantics of OR-joins has to be defined carefully without using future states, i.e., the state space.

A second problem is that two or more OR-joins may mutually depend on each other as one OR-join can be executed only if the other is not and vice versa. These situations are called *vicious circles* [3] and have high relevance in practice [2].

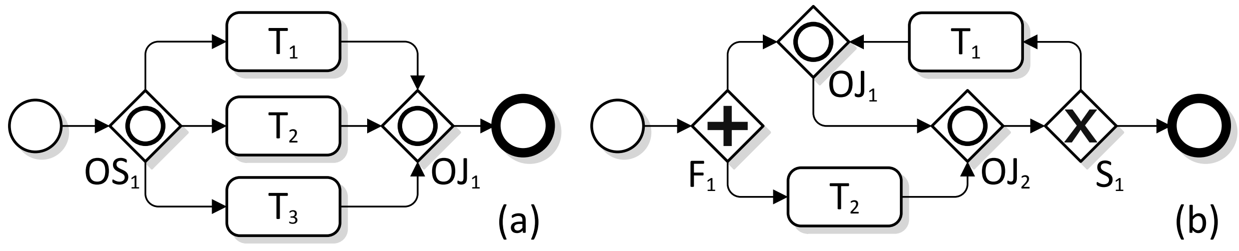


Figure 1. Two processes in BPMN notation containing OR-joins

The right-hand side *(b)* of Figure 1 addresses the introduced difficulties (following [2]). It contains a diverging AND gateway (a fork) F_1 supplying both OR-joins OJ_1 and OJ_2. The simple semantics of the previous example does not hold since the OR-join OJ_1 would wait for the execution of OJ_2 and vice versa — a vicious circle. Such vicious circles are well-researched and there are a lot of well-established and carefully defined semantics of OR-joins, e.g., [2], [3], [4], [5], [6], [7], [8], [9], [10], with a considerable practical relevance.

However, Figure 2 illustrates a sample process which leads to failures by using state-of-the-art semantics. It describes a car service in a garage. At first, the garage receives a car for an inspection and creates an order. Afterwards, an accounting system creates the corresponding invoice and, at the same time, the inspection starts. During the inspection, the customer may have special requests, e.g., changing the oil or testing the air pressure of the tires. Those special requests would be performed at the same time like the inspection. If all requests are done, the garage cleans up the car as a special service, followed by an inspection of the customer. If the customer is unsatisfied about the cleanness of his or her car, the garage keeps cleaning the car until the customer is pleased.

If we throw away all textual information and eliminate the special request part from the sample process, we get the same abstract process as illustrated on the right-hand side of Figure 1 *(b)*. We will use that abstract process as counter example in Section 4 where we show that it leads to failures (and therefore cannot be properly executed) by the use of state-of-the-art semantics.

For this reason, in this paper, we provide a general definition of *soundness* for business processes which promises the absence of failures [11], [12], [13]. Our common definition of soundness makes it possible to define *completeness*, building a criterion to evaluate the quality of existing OR-join semantics with regard to the number of sound processes being successfully executable. Therefore, we will show that all existing OR-join semantics (to the best of the authors' knowledge), especially the popular approach of Völzer [2], are good approximations of a complete OR-join semantics — however none is complete, i.e., they lead to failures in sound processes as mentioned before.

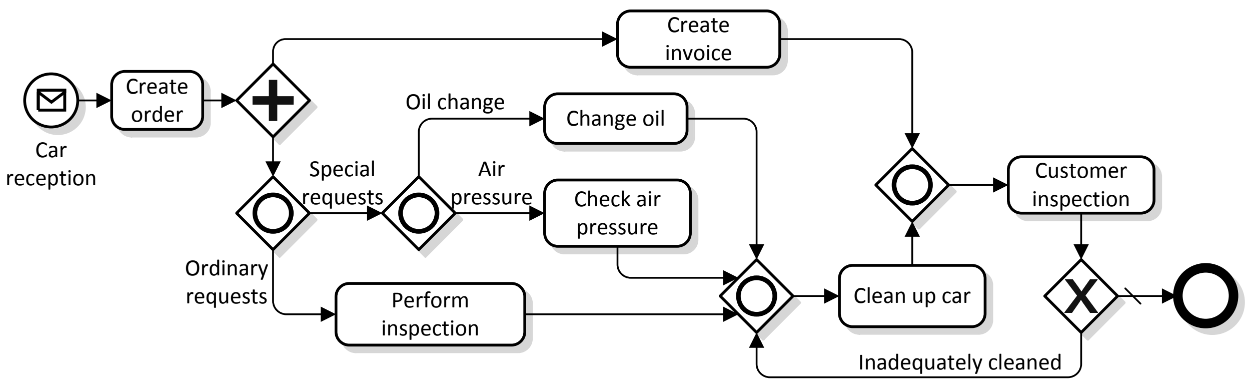


Figure 2. Car service process

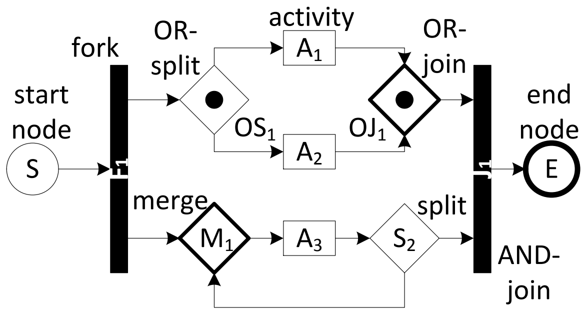


Figure 3. A workflow graph with OR-joins

The definition of a (first) complete OR-join semantics as described in this paper has several advantages: 1) The definition of an intuitive and correct semantics for process modelers is simpler if the limits of OR-join semantics are known. 2) A successful replacement of OR-joins with converging XOR gateways (merges) or converging AND gateways (AND-joins) needs completeness to be correct. 3) Although someone may argue for the poor modeling of processes with OR-joins in cases where state-of-the-art semantics do not work, it is always possible that accurately modeled business processes are not correctly executable when using OR-joins in process collaborations, in big processes, or for reasons of readability and compactness of resulting processes.

The rest of the paper is structured as follows: Section 2 summarizes some definitions and notions, followed by the definition of soundness and completeness with OR-joins in Section 3. Afterwards, we evaluate existing OR-join semantics with regard to completeness (Section 4) and introduce a first complete semantics in Section 5. The complete semantics will then be applied on two examples in Section 6. Finally, Section 7 concludes this paper.

2 Preliminaries

In this section, we introduce some basics of processes. Processes are mostly represented as workflow nets (Petri nets) [12] or workflow graphs being special cases of control flow graphs allowing explicit parallelism [11]. We prefer workflow graphs since the modeling of OR gateways in Petri nets is difficult [10]. Before we define workflow graphs, the term of *paths* is introduced. Formally, a *path* $P = (n_1, \ldots, n_{m-1}, n_m), m \geq 2$, for a digraph $G = (N, E)$ is a sequence of nodes of N such that $\forall i \in \{1, \ldots, m-1\} : (n_i, n_{i+1}) \in E$. We write $n \in P$ iff $n \in \{n_1, \ldots, n_m\}$. Sometimes, we use a similar definition of paths consisting of edges and depending on the context, i.e., a *path* $P = (e_1, \ldots, e_{m-1}, e_m), m \geq 2$, is a sequence of edges of E such that $\forall i \in \{1, \ldots, m-1\}$: the target node of e_i is the source node of e_{i+1}.

A *workflow graph* is a directed graph $WFG = (N, E)$ where N is a set of nodes which is partitioned into disjoint sets of *activities* N_A, *forks* N_F, *AND-joins* N_J, *splits* N_S, *merges* N_M, *OR-splits* N_{OS}, *OR-joins* N_{OJ}, and $\{start, end\}$ such that (1) $start$ is the unique start node and end is the unique end node, (2) the end node, activity, split, OR-split, and fork each have exactly one incoming edge, (3) merges, OR-joins, and AND-joins have at least two incoming edges, (4) the start node, activity, merge, OR-join, and AND-join each have exactly one outgoing edge, (5) splits, OR-splits, and forks have at least two outgoing edges, and (6) each node $n \in N$ lies on a path from the start node to the end node.

Figure 3 shows a sample workflow graph. The start and end nodes are depicted as (thick) circles, whereas an activity is depicted as a rectangle. Forks and AND-joins are illustrated as thin massive rectangles, and splits and merges as (thick) diamonds. OR-splits and OR-joins are depicted as (thick) diamonds with dots in their center.

Usually, the semantics of workflow graphs is defined, similarly to Petri nets, as a token game. Therefore, a *state* of a workflow graph $WFG = (N, E)$ is represented by tokens on the edges of the graph, i.e., a state of WFG is a mapping $S\colon E \to \mathbb{N}$ which assigns a natural number to each edge. If $S(e) = k$, we say that edge e carries k tokens in state S. In the *initial state*, only the outgoing edge of the start node carries a token, whereas in the *termination state*, only the incoming edge of the end node carries a token. After a node $n \in N$ will be executed (fired) in state S, the state changes into a state S', depicted as $S \overset{n}{\to} S'$. A state S' is *reachable* from a state S (written $S \to^* S'$) if there is a finite sequence $S \overset{n_1}{\to} S_1 \ldots S_{k-1} \overset{n_k}{\to} S'$, $k \geq 0$.

Basically, a semantics of an arbitrary node n can be divided into two parts: The *enabledness* of n in a state S, i.e., whether node n can be executed or not, and the *effect* on state S if n will be executed. The effect is traditionally defined as a modification of the number of tokens on the incoming and outgoing edges of the executed node. An activity, a fork, and an AND-join remove one token from each of their incoming edges and add one token to each of their outgoing edges. A merge removes non-deterministically one token from one of its incoming edges carrying a token and adds one token to its outgoing edge. In workflow graphs that carry no data as, referred in this paper, a split removes one token from its incoming edge and decides non-deterministically on which of its outgoing edges it will add one token. An OR-split performs the same way as a split; however it adds one token for each edge of a non-empty subset of its outgoing edges. At this moment, it is assumed that an OR-join removes one token of all of its incoming edges carrying a token and adds one token to its outgoing edge. Vanhatalo et al. [14] formalize the effect of a node n and a state change $S \overset{n}{\to} S'$ (also for OR-joins) in general as follows:

1. n is a fork, AND-join, or an activity:

$$S'(e) = \begin{cases} S(e) - 1, & e \text{ is an incoming edge of } n \\ S(e) + 1, & e \text{ is an outgoing edge of } n \\ S(e), & \text{otherwise} \end{cases}$$

2. n is a split or merge and there are an incoming e' edge with $S(e') \geq 1$ and an outgoing edge e'' of n:

$$S'(e) = \begin{cases} S(e) - 1, & e = e' \\ S(e) + 1, & e = e'' \\ S(e), & \text{otherwise} \end{cases}$$

3. n is an OR-split or OR-join and there is a non-empty set Out of outgoing edges of n:

$$S'(e) = \begin{cases} S(e) - 1, & e \text{ is an incoming edge of } n \text{ such that } S(e) \geq 1 \\ S(e) + 1, & e \in Out \\ S(e), & \text{otherwise} \end{cases}$$

Thus, tokens *(may) travel* through the workflow graph depending on the nodes (possibly) firing. Moreover, we say that a token on an edge e in state S *arrives* on an incoming edge of a node n (or simply arrive at n) iff the token of e travels through state changes to that incoming edge of n.

The enabledness of a node can be seen as a mapping from a node n and a state S to *enabled* or *not enabled*, called *release function*. In general, if $\mathfrak{S}$ is the set of all possible states of WFG, the mapping is defined as $F : N \times \mathfrak{S} \to \{\text{ enabled, not enabled }\}$. To keep the release function simple, we describe only the conditions when the release function evaluates to *enabled* in the remainder of this paper, knowing that it evaluates to *not enabled* otherwise. Basically, a node is *not* enabled in a state S if it has no incoming edge which carries a token. In the following, we call each node n *active* in state S if this basic property (that one of n's incoming edges carries a token) holds. Most nodes (activities, splits, merges, forks, and OR-splits) are actually enabled if they are active. However, an AND-join is not enabled until each of its incoming edges has a token. Since the release function of OR-joins is the focus of this paper, we refer to it simply as *release function* in the following and introduce some approaches in Section 4.

3 Soundness, Completeness, and Liberty

Soundness, introduced by van der Aalst [15], is one of the most frequently used correctness criterion of *business* processes and guarantees the absence of deadlocks and lack of synchronization if data is not considered. A state S of WFG is a *deadlock* iff there is a token on an incoming edge of a node n such that each reachable state from S carries also a token on that edge. We also say that n causes a deadlock. S has a lack of synchronization iff there is an edge which carries more than one token in S. Usually, a workflow graph is called *sound* iff no deadlock or state with a lack of synchronization is reachable from the initial state.

Since the release function of OR-joins is currently undefined, we cannot compute reachable states from an initial state. Therefore, we define soundness regarding OR-joins in its most general manner:

Definition 1 (Soundness). *Let $WFG = (N, E)$ be a workflow graph.*

WFG is sound *iff there is a release function F of OR-joins such that from the initial state neither a deadlock nor a state with a lack of synchronization is reachable. In this case, we say that F holds for WFG.*

This definition has the following advantages:

(1) The definition is equal to classical soundness iff the workflow graph has no OR-joins since each arbitrary release function can be used.
(2) If the workflow graph is unsound without regarding the release function, it is unsound for each arbitrary release function.
(3) If and only if there is a sound (non-deterministic) execution of the workflow graph, there is at least one valid release function.

We have a look at the four workflow graphs of Figure 4 as examples for this general definition of soundness. The workflow graph *(a)* is sound as there exists a simple release function F for the OR-joins that does not result not in a deadlock nor in a lack of synchronization, e.g., waiting for all tokens on those edges which have a path to it. The workflow graph *(b)* is naturally not sound as there is no release function that can avoid the possible lack of synchronization in the merge M_1, above all as the workflow graph has no OR-joins. Workflow graph *(c)* is not sound either and workflow graph *(d)* is sound. The reason why the last both workflow graphs are (not) sound is explained in the following, especially, in Section 5.

With the help of this general definition of soundness, we can define a criterion — completeness — to evaluate the quality of a release function.

Definition 2 (Completeness). *Let $\mathfrak{WFG}$ be the set of all* sound *workflow graphs and $\mathfrak{F}$ the set of all release functions holding for each $WFG \in \mathfrak{WFG}$.*

A release function F is complete *iff $F \in \mathfrak{F}$. We say that a complete release function is a* most liberal *one.*

4 Evaluation

In the last section, we have defined the notion of completeness which allows the evaluation of existing OR-join semantics approaches with regard to their liberalness. For this, we introduce a sample workflow graph (cf. Figure 5) being used as a counterexample to show that existing release functions are not as liberal as possible. If this workflow graph will be executed, it starts its execution with a single token on the incoming edge of the fork F_1. Since that fork is then enabled, it fires and the activity A_2 can be executed so that the OR-joins O_1 and O_2 have a token on exactly one of their incoming edges e_1 and e_3 in a state S. Now, there are four possibilities in S:

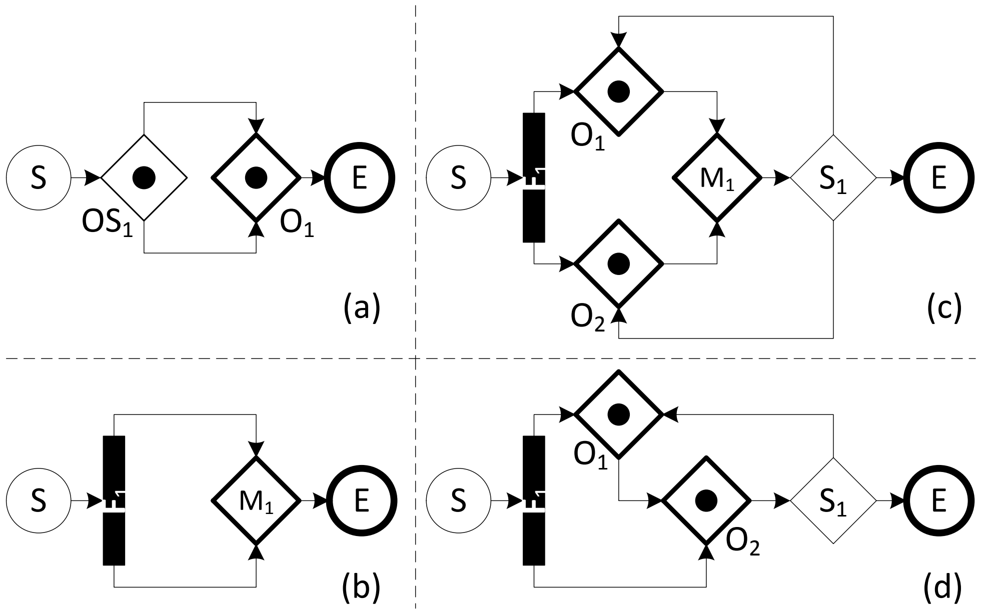


Figure 4. Sound and unsound workflow graphs

(1) No OR-join is enabled: S would be a deadlock. $\lightning$

(2) OR-join O_1 is enabled and O_2 is not. As a consequence, O_1 fires and O_2 gets a token via its upper incoming edge e_4. Therefore, O_2 synchronizes both tokens on its incoming edges to a single one on its outgoing edge. As a result, no deadlock and no state with a lack of synchronization is actually possible afterwards. ✓

(3) OR-join O_2 is enabled and O_1 is not. However, after O_2 has been fired, the split S_1 may decide to put a token on the incoming edge e_8 of the end node. If the other token eventually reaches e_8 (which is the case since the termination state of this workflow graph must have only one token on e_8), each termination state has a lack of synchronization. $\lightning$

(4) Both OR-joins are enabled and fire simultaneously. However, this concurrent execution may result in the same situation as in possibility *(3)*. $\lightning$

In summary, only possibility *(2)* results in a sound behavior. There is a (yet undefined) release function for which our example is sound by Definition 1.

A possible release function $F_{example}$ holding for that example evaluates to *enabled* in a state S for one of the both OR-joins j iff

1. there is at least one incoming edge of j which carries a token in S and
2. a. if $j = O_1$ and none of the edges e_0,e_5,e_6, and e_8 carry a token in S
   b. if $j = O_2$ and none of all edges except e_3 and e_4 carry a token in S.

We compare three popular OR-join release functions in the following. For this, we assume an OR-join j and a state S of an arbitrary workflow graph WFG for the definitions of release functions.

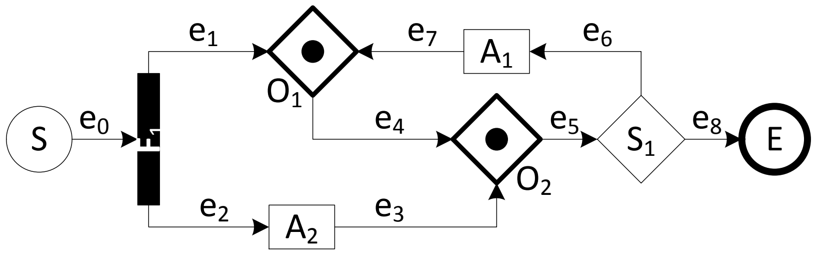


Figure 5. Workflow graph of the process in Figure 1 *(b)*

At first, we start with a state-space based approach of Kindler [3] that essentially considers all reachable states. As mentioned before, to consider all reachable states, we have to know the release function of OR-joins. However, to define this release function, we have to consider all reachable states. This is a cyclic dependency and it prevents a straight-forward formalization of the release function. Kindler solved this problem with the fix-point theory. At this point, we accept that Kindler has found a mathematical sound formalization. Then, the release function $F_{Kindler}$ evaluates to *enabled* iff

1. at least one incoming edge of j carries a token in S and
2. there is no token except the tokens on j itself which can travel in a reachable state of S to an incoming edge of j that carries no token in S.

With this in mind, we see that our counter example leads to a deadlock since the token of O_1 may travel to O_2 if O_1 fires first or vice versa (cf. possibility *(1)*). Therefore, the semantics of Kindler cannot be complete.

Dumas et al. [6] follow a graph-based approach. They define their release function recursively. It evaluates to *enabled* for j in S in simple terms iff j has an incoming edge which carries a token in S and none of j's direct predecessor nodes, e.g., p, without a token on the edge (p, j) has a (indirect) predecessor which is enabled. To avoid arbitrary recursion depths, each second enabledness evaluation of an indirect predecessor node p' of p always evaluates to *not enabled*. Therefore, the release function F_{Dumas} contains an additional set V of visited nodes. $F_{Dumas}(j, S, V)$ evaluates to *enabled* with the function call $F_{Dumas}(j, S, \emptyset)$ iff

1. $j \notin V$ and there is an incoming edge e of j with $S(e) \geq 1$ and
2. for each edge $e' = (n, j)$ with $S(e') = 0$ exists no predecessor p of n for which $F_{Dumas}(p, S, V \cup \{j\})$ is *enabled*

Every time two OR-joins cannot be enabled until the other is not (cf. our counter example), both are treated as enabled. Thus, they are executed non-deterministically such that O_1 will be executed before O_2 or O_2 will be executed before O_1. That may result in a state with a lack of synchronization (cf. possibility *(3)* and *(4)*). For this reason, F_{Dumas} is not complete.

Christiansen et al. [9] directly follow the BPMN 2.0 specification [1], i.e., they follow the approach of Völzer [2] as it is part of the current BPMN specification. The release function F_{Volzer} evaluates to *enabled* for j in state S iff

1. there is an incoming edge e of j with $S(e) \geq 1$ and
2. each other edge e' of WFG with $S(e') \geq 1$ having a path to an incoming edge of j has also a path to e without passing j

The argumentation of Völzer is that — in simple terms — if there is a possibility that a token reaches an incoming edge e of j which already carries a token, then j should be fired first to avoid a lack of synchronization. Although this argumentation is comprehensible, our sound workflow graph in Figure 5 would lead to a deadlock since the token of O_1 can travel to the tokenless incoming edge e_4 of O_2 and vice versa. Therefore, both OR-joins are not enabled. Völzer himself has already named that effect in his paper.

In summary, no considered OR-join release function is complete and therefore nobody can execute all sound workflow graphs successfully — there is the need for a most liberal release function.

5 A Complete OR-Join Semantics

In the remainder of this section, we handle AND-joins as a special case of OR-joins for which tokens always arrive at all incoming edges. This is possible in sound workflow graphs as AND-joins cause no deadlocks. For this, we refer to all OR- and AND-joins as OR-joins.

In the former sections, we have evaluated existing release functions. Since there exists no complete one, we want to derive our own approach for the first complete theory. Hence, we start at a common property of most release functions: An OR-join j with a token on one of its incoming edges is not enabled since it has to "wait" for a further event — in general that no more tokens can ever arrive at an incoming edge of j. We formulate this "waiting" of an OR-join as follows:

Definition 3 (Waiting). *Let $WFG = (N, E)$ be a sound workflow graph using a release function F, j be an OR-join, and e be not an incoming edge of j.*

j waits *for e in state S iff*

1. *j is active,*
2. *e carries a token, and*
3. *there is at least one path P from e to an incoming edge of j, such that no node touched by P prevents the token on e from travelling via P to j (without firing j).*

j waits for node n iff e is an incoming edge of n. j always *waits for e iff in each state, where j is active and e carries a token, j waits for e. j* never *waits for e iff in each such state j does not wait for e.*

Actually, this definition is very general: An OR-join j will not be reached by a token on an edge e in a state S for two cases: Either (1) there is actually no path from e to an incoming edge of j or (2) on all these paths, the token would be prevented from traveling to j. It is simple to determine whether there is a path from e to an incoming edge of j. The interesting nodes are those which prevent a token from travelling on a path from one edge to another edge. In the following, we introduce the basic property of nodes *always* preventing the traveling of tokens. As a consequence of this property, such nodes can only be OR-joins and they can only prevent the travel of tokens within cyclic workflow graphs. In such graphs, there are sometimes situations in which OR-joins seem to mutually wait for each other and therefore would cause a deadlock. With this in mind, we subsequently define an appropriate execution order for such situations deciding which OR-joins should wait and which OR-joins *must* not wait. As a result of these considerations, we derive our approach for a complete release function which we prove to be complete in conclusion.

5.1 Basic Property

As mentioned before, we introduce the basic property of a node n always preventing the traveling of tokens from an edge e to an incoming edge of an OR-join j. If such a node n exists on all possible paths from e to an incoming edge of j, j does never have to wait for e. The following lemma describes that property which holds in this respect for sound workflow graphs.

Lemma 1. *Let $WFG = (N, E)$ be a sound workflow graph, j be an active OR-join in a state S, and e be an edge carrying a token in S.*

j never waits for e

$$\Longrightarrow$$

On all paths from e to an incoming edge of j without passing j,
an OR-join always waiting for j can be activated

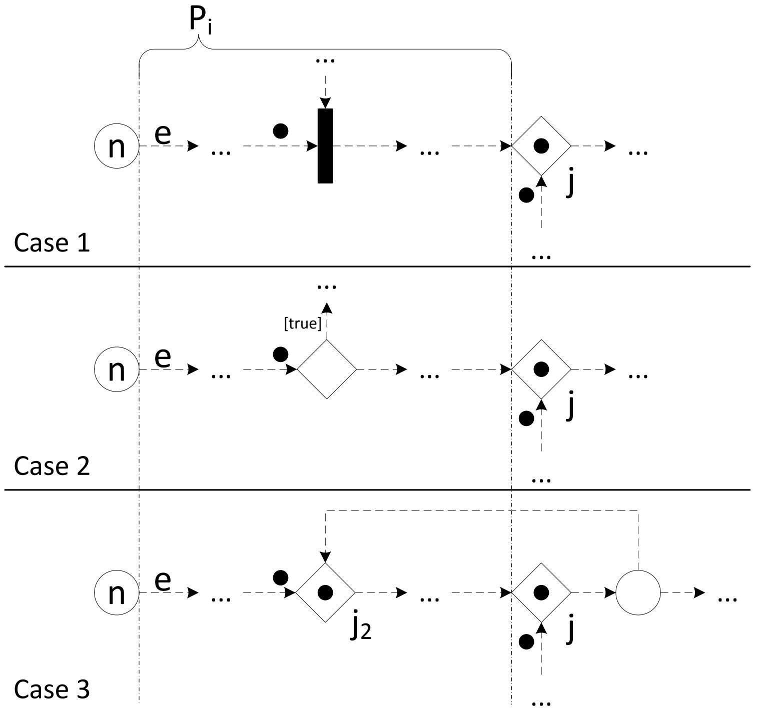


Figure 6. Cases how a token can never arrive at j

Proof (Lemma 1). There are two possibilities of j never having to wait for e:

1: There is not a path from e to an incoming edge of j without passing j. ✓

2: There are paths from e to an incoming edge of j without passing j.
Let P_i be one of those paths $\overset{\text{Def. 3}}{\Longrightarrow}$ a token from e can never arrive at j via P_i in a reachable state since otherwise j would wait for $e \Rightarrow$ the token either (1) is blocked on P_i, (2) always leaves P_i, or (3) activates a node on P_i which always waits for j (cf. Figure 6).
Case 1: The token is blocked on $P_i \Rightarrow$ a deadlock must have occurred. $\lightning$
Case 2: The token always leaves $P_i \Rightarrow$ there is a split or an OR-split on P_i which always decides that the token does not travel on P_i. Since splits and OR-splits decide non-deterministically, this case is impossible. $\lightning$
Case 3: The token activates a node on P_i which always waits for j. This node must be an OR-join j_2 since it must be able to wait. Since there would be a deadlock if j also waits for j_2, j never has to wait for j_2. ✓ □

In the next lemma, we show that the inversion holds too.

Lemma 2. *Let $WFG = (N, E)$ be a sound workflow graph, j be an active OR-join in a state S, and e be an edge carrying a token in S.*

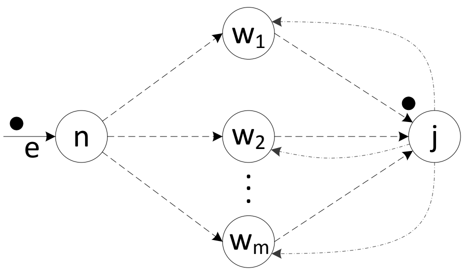
On all paths from e to an incoming edge of j without passing j,
an OR-join always waiting for j can be activated

$$\Longrightarrow$$

j never waits for e

Proof (Lemma 2). Constructive proof.

Case 1: There is no a path from e to an incoming edge of j without passing $j \overset{\text{Def. 3}}{\Longrightarrow} j$ never waits for e ✓



Case 2: There are paths from e to an incoming edge of j without passing j (see the figure on the right of this text) $\overset{\text{assumption}}{\Longrightarrow}$ if the token on e travelled a path P_i to j, it would have activated an OR-join w_i which would always wait for j
$\overset{\text{deadlock-free}}{\Longrightarrow}$ j never waits for w_i
$\Longrightarrow j$ can always be executed before w_i
$\Longrightarrow j$ never waits for e ✓ □

Then, the following theorem formulates our basic property:

Theorem 1. *Let $WFG = (N, E)$ be a sound workflow graph, j be an active OR-join in a state S, and e be an edge carrying a token in S.*

j never waits for e

$$\Longleftrightarrow$$

On all paths from e to an incoming edge of j without passing j,
an OR-join always waiting for j can be activated

Proof (Theorem 1). Follows directly from Lemma 1 and 2. □

5.2 Determination of an Appropriate Execution Order

In the case of acyclic sound workflow graphs, the definition of a release function is simply a result of the previous basic property: An OR-join j only waits in state S for all those edges carrying a token and having a path to an incoming edge of j. Otherwise, there must be a cycle.

In contrast, in the cyclic case, two OR-joins may mutually wait for each other and therefore may cause a deadlock: A *vicious circle* [3]. In order to avoid these circles, the definition of a release function has to describe in principle the appropriate execution order deciding which OR-join in such circles has to wait and which OR-join must not wait. A concept of compiler theory helps to find an important property in which an OR-join *always waits* for a token on an edge — the post-dominance relation.

Definition 4 (Post-Dominance). *Let $WFG = (N, E)$ be a workflow graph with its end node* end.

A node n post-dominates *node m if every path from m to* end *contains n, written n pdom m. n* properly *post-dominates m if n pdom m and $n \neq m$. n* directly *post-dominates m if n properly post-dominates m and any other node o that properly post-dominates m also properly post-dominates n, i.e., the direct post-dominance relation is transitive.*

A post-dominator tree *of WFG is a directed graph $pdom(WFG) = (N, E')$ with $E' = \{(n, m) : n$ directly post-dominates m in $WFG\}$.*

Figure 7 shows the sample workflow graph *(a)* of Figure 5 and its post-dominator tree *(b)*.

In sound workflow graphs, the post-dominance relation guarantees that a post-dominator j of an active node n will be executed in each case. Hence, such a token on an incoming edge of n travels eventually on a path to the incoming edge of the end node (and so via j) to be consumed in the termination state. The next theorem uses this fact and describes an important property in which an OR-join *always waits* for nodes which it post-dominates.

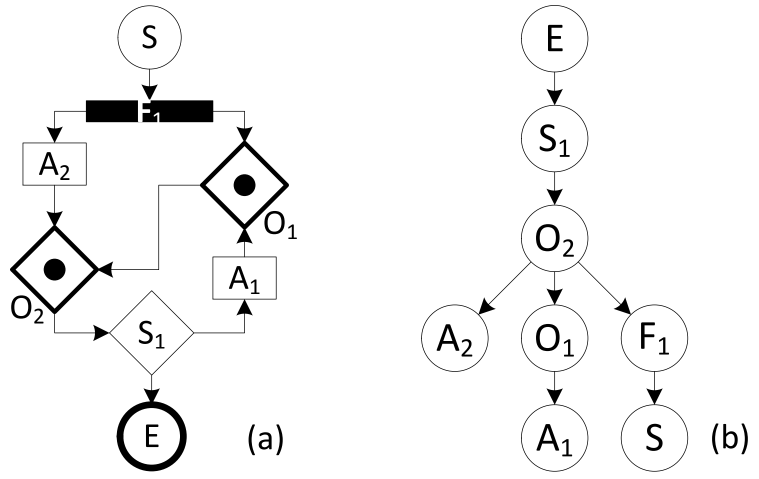


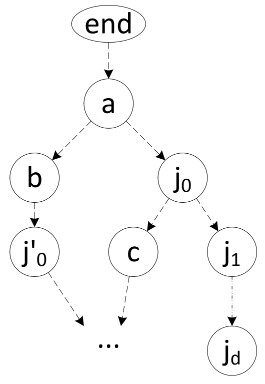
Figure 7. The workflow graph *(a)* of Figure 5 and its post-dominator tree *(b)*

Theorem 2. *Let $WFG = (N, E)$ be a sound workflow graph, j be an active OR-join in a state S, and n another active node in S.*

$$j \textit{ lies on all paths from } n \textit{ to the end node } (j\ pdom\ n)$$
$$\Longrightarrow$$
$$j \textit{ always waits for } n$$

Proof (Theorem 2). Let $pdom(WFG)$ be the post-dominator tree of WFG. It defines a partial order caused by the post-dominance relation.

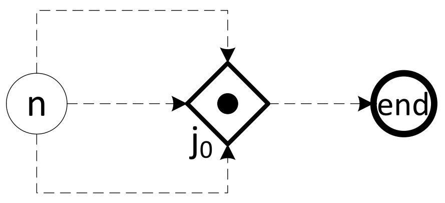
We use this order to assign numbers to each OR-join (cf. the figure on the right of this text): Each OR-join j is given the number of OR-joins lying on the path from j to the root node, e.g., an OR-join labelled d has d predecessors being OR-joins in $pdom(WFG)$. We write j_d for an OR-join j with d predecessors being OR-joins in $pdom(WFG)$. We can reformulate the hypothesis of this lemma as follows if j_d is an active OR-join in state S and n is another active node in S:



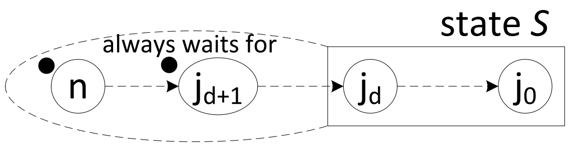
$$j_d\ pdom\ n \quad \Longrightarrow \quad j_d \text{ always waits for } n$$

This proposition will be proven by mathematical induction and by contradiction. Therefore, we start with $d = 0$.

Assumption: j_0 does not have to wait for n in a state S
$\Rightarrow$ j_0 can be executed independently from n in a state S_1, $S \rightarrow^* S_1$.
Assume j_0 is executed before n in S_1
$\Rightarrow$ the end node can be activated in a subsequent state S_2, $S_1 \rightarrow^* S_2$, since there is a path from j_0 to end without another OR-join
$\Rightarrow$ end and n are active in S_2
$\Rightarrow$ S_2 has a lack of synchronization when the token from the outgoing edge of n eventually arrives at the incoming edge of end. ↯

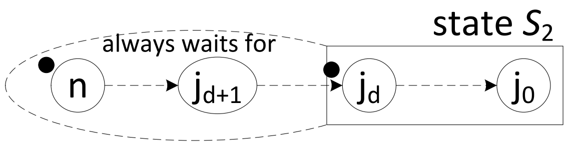


The induction step is based on the hypothesis being valid for j_d. So we have to show that it holds for j_{d+1}. The figure on the righthand side reflects the overall situation.

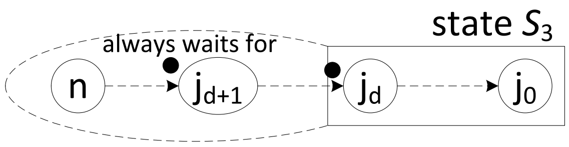


Assumption: j_{d+1} does not have to wait for n in a state S
$\Rightarrow j_{d+1}$ can be executed independently from n in a state S_1, $S \to^* S_1$.
Assume j_{d+1} is executed before n in S_1
$\Rightarrow j_d$ can be activated in a following state S_2, $S_1 \to^* S_2$ (since there is a path P from j_{d+1} to j_d without another OR-join).

Assume j_d is activated in S_2 and n has not fired yet
$\overset{\text{hypothesis}}{\Rightarrow}$ j_d always waits for n
$\overset{\text{deadlock-free}}{\Rightarrow}$ n never waits for j_d
$\Rightarrow$ the token on the outgoing edge of n can arrive at j_{d+1} in a state S_3, $S_2 \to^* S_3$, after n has been fired before j_d.



Assume j_{d+1} is again activated and it is active at the same time as j_d in S_3
$\overset{\text{hypothesis}}{\Rightarrow}$ j_d always waits for j_{d+1} since j_d post-dominates j_{d+1}
$\overset{\text{deadlock-free}}{\Rightarrow}$ the same path P from j_{d+1} to j_d can be traveled once more by the token of the outgoing edge of j_{d+1} after firing j_{d+1}
$\Rightarrow$ there is a state S_4, $S_3 \to^* S_4$, with a lack of synchronization at an incoming edge of j_d. $\lightning$ $\square$



As a result, the following corollary is directly derived from the previous theorem since two OR-joins mutually waiting for each other would cause a deadlock.

Corollary 1. *Let $WFG = (N, E)$ be a sound workflow graph, j be an active OR-join in a state S, and n another active node in S.*

$$n \textit{ is post-dominated by } j$$
$$\Longrightarrow$$
$$n \textit{ never waits for } j$$

In principle, Corollary 1 shows that in vicious circles, an OR-join does not have to wait for its post-dominator OR-joins. For now, we assume that vicious circles have only an appropriate execution order if one OR-join post-dominates the other. The correctness of this assumption, i.e., the completeness of our release function, will be proven afterwards in Theorem 3. Under these terms, the following definition that results, in particular, from the previous corollary describes our approach of a release function of an OR-join.

Definition 5 (Release Function). *Let $WFG = (N, E)$ be a sound workflow graph and j be an OR-join.*

The release function F_{compl} evaluates to enabled *for j in a state S iff*

1. *there is an incoming edge e of j with $S(e) \geq 1$ and*
2. *on each path P without passing j from an active node n in S to j lies an OR-join which post-dominates j*

As a result of our property (Theorem 1), this definition implicitly makes use of the fact that an active OR-join j in a state S never waits for an edge e carrying a token if there is an OR-join post-dominating j on all paths from e to that j.

Since we now have defined our approach of a release function, we can apply it to our counter example (cf. Figure 5) of Section 4. The situation is the same: Both OR-joins O_1 and O_2 are active in a reachable state S from the initial state. If we apply our semantics, OR-join O_2 waits for O_1 and O_1 is enabled to fire since O_2 post-dominates O_1. It results in the desired behavior of possibility *(2)* (cf. Section 4) describing that O_2 will be executed before O_1 if both are active in the same state. In this case and as mentioned before in Section 4, this execution order leads to a sound behavior.

5.3 Completeness of our Approach

Our assumption was that vicious circles only can be resolved if one of the OR-joins post-dominates the other. If that holds, our release function would be complete. In the following theorem, it will be proven that this assumption is correct, i.e., there is no appropriate execution order otherwise.

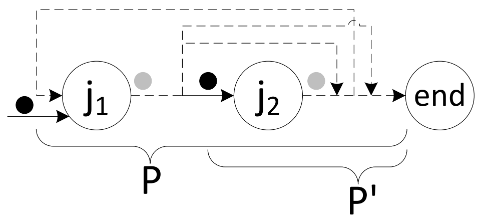
Theorem 3 (Completeness). *Let $WFG = (N, E)$ be a workflow graph by using an arbitrary release function F and j_1, j_2 be two different active OR-joins in a state S which have at least one path to one another.*

$$j_1 \textit{ does not post-dominate } j_2 \textit{ and } j_2 \textit{ does not post-dominate } j_1$$
$$\Longrightarrow$$
$$WFG \textit{ is unsound}$$

Proof (Theorem 3). Constructive proof.

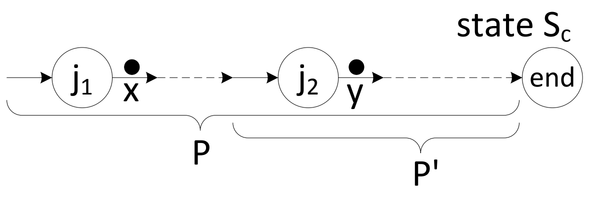
Case a: j_1 always waits for j_2 and j_2 always waits for j_1 — a deadlock. ✓

Case b: j_1 does not wait for j_2 or j_2 does not wait for j_1 in state S. Without loss of generality, we assume that j_2 does not wait for $j_1 \Rightarrow j_2$ is executed before or at the same time as j_1. Furthermore, there is at least one path P' from the incoming edge of j_2 carrying a token in S to the incoming edge of the end node without an incoming edge of j_1 (since j_1 does not post-dominate j_2). There is also one path P from the incoming edge of j_1 carrying a token in S to the incoming edge of the end node where P contains P' as a subpath since there is a path from j_1 to j_2 (cf. conditions of this Theorem 3). The figure above illustrates the situation conceptually.



Since j_2 does not wait for j_1 in state S as mentioned before, let j_2 be executed before or at the same time as j_1 in a reachable state S_1. As a consequence, in both cases, j_2 is executed and its outgoing edge carries a token in S_1. A possible state S_1 of an execution, which, e.g., arose from a concurrent execution of j_1 and j_2, is illustrated in the figure above with grey dots.

Now, we assume a variable S_c for the current state (initially S_c is S_1) and a variable x which initially represents the incoming or outgoing edge of j_1 (depending on whether j_1 was executed before state S_1 or not) and which describes the position on path P of the "token of j_1" in the current state S_c. Furthermore, there is a variable y which initially is the outgoing edge of j_2 and which represents the position on the *same* path P of the "token of j_2" in S_c.



In S_c, there always exists two possibilities for the nodes n_x and n_y being currently activated by the edges $x = (n', n_x)$ and $y = (n'', n_y)$:

Case 1: n_x cannot be executed and n_y cannot be executed since n_x always waits for n_y and n_y always waits for $n_x \Rightarrow$ deadlock

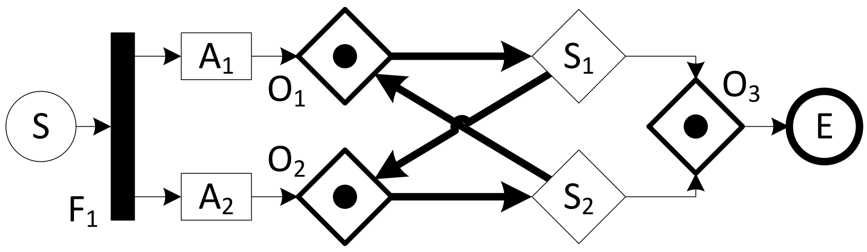
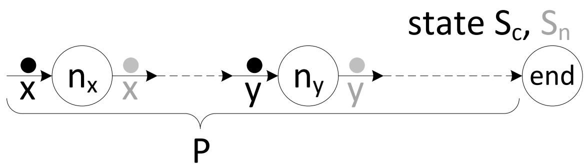


Figure 8. A workflow graph with two OR-joins mutually waiting for each other



Case 2: n_x can be executed or n_y can be executed in S_c (possibly at the same time) $\Rightarrow$ n_x or n_y will be executed, tokens could be put on that outgoing edges of n_x (if n_x was executed) or n_y (if n_y was executed) which lie on path P, the followed state is S_n $\Rightarrow$ afterwards, let S_c be S_n and x and y those edges on P now carrying a token, i.e., the mentioned outgoing edges of n_x and n_y.

Both cases show that there is either a deadlock or both tokens can exclusively travel on possibly different edges x and y of path P. Since both tokens on x and y must eventually arrive at the same edge of path P' in future ($x = y$), this would lead to a state with a lack of synchronization. All cases results in an unsound behavior. ✓

In all cases, an execution could result in a deadlock or in a state with a lack of synchronization. Hence, the workflow graph must be unsound! ✓ □

In summary, our proposed release function fits in acyclic cases since active OR-joins always have to wait for tokens on edges which have paths to them (Theorem 1). In the cyclic case, only vicious circles prevent a simple definition of a release function as deadlocks occur (caused by mutually waiting OR-joins). Such situations can only be resolved if one OR-join post-dominates the other as mentioned in Theorem 3 — otherwise, a deadlock or a state with a lack of synchronization is possible. In short, our approach must be complete as all cases are handled.

The following example shows a workflow graph which has a deadlock caused by two mutually waiting OR-joins — also by using our complete release function. Figure 8 shows this workflow graph with a vicious circle $O_1 \rightarrow S_1 \rightarrow O_2 \rightarrow S_2 \rightarrow O_1$ and demonstrates that deadlocks actually are not exclusive phenomenons of AND-joins: After the execution of F_1, both OR-joins O_1 and O_2 are activated and therefore, O_1 waits for O_2 and vice versa since neither O_1 post-dominates O_2 nor O_2 post-dominates O_1 and there are paths to each other. This state is a deadlock. As stated in Theorem 3, the workflow graph is unsound. For example, if we allow one or both of the OR-joins to be executed, the two tokens may arrive at the same incoming edge of the OR-join O_3.

6 Semantics in Practice

After introducing and proving our complete release function — and therefore the most liberal semantics of OR-joins —, we show how that semantics can be applied in practice. For this purpose, we use the realistic sample process of Figure 2 as shown once again with labels in Figure 9.

We imagine a reachable state S in which the edges e_4,e_{15}, and e_{16} carry a token as illustrated in the figure with black dots. One token (on e_4) activates the OR-join OJ_2, another token on e_{15} activates the OR-join OJ_1. Existing OR-join semantics now may lead to a deadlock or to a state with a lack of synchronization as mentioned in Section 4. Applying our semantics, however, makes the process in each case executable in failure-free way. Therefore, at first, we determine the post-dominator tree of our sample process (cf. Figure 10).

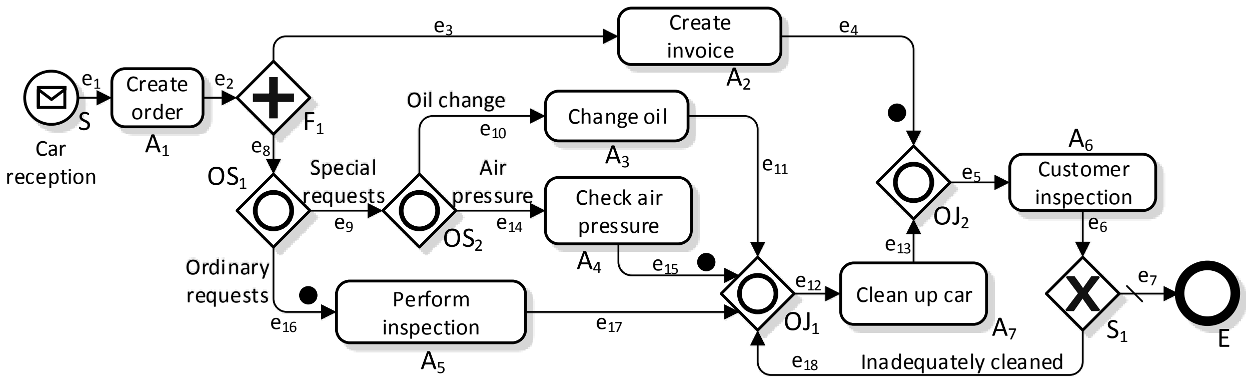


Figure 9. Car service process with labels

The tree shows that OR-join OJ_2 post-dominates OJ_1. Therefore, on each path from the edge e_4 to an incoming edge of the OR-join OJ_1, OR-join OJ_2 is passed, i.e., OJ_1 does not wait for the token of the edge e_4 (cf. Def. 5). As a result, OJ_1 would wait for possible tokens on the edges e_1, e_2, e_5, e_6, e_8, e_9, e_{10}, e_{14}, e_{16}, and e_{18}; and therefore waits in the current state S for the token of e_{16}. On the other hand, OJ_2 waits for all edges currently carrying a token as there is no OR-join that directly post-dominates OJ_2, i.e., OJ_2 would wait for possible tokens on all edges except e_4, e_7, and e_{13}. As we have proven, the process would never produce a failure situation and leads to a sound behavior.

The explanation of the previous example shows furthermore that the set of edges $\omega(j)$ an active OR-join j would wait for when one of those edges carries a token can be precomputed before an execution of a process. If we assume that we use a state representation which is a subset of the set of edges carrying a token, then the evaluation whether an active OR-join j can be executed or not in a state S is possible in constant time at runtime ($S \cap \omega(j) = \emptyset$).

Another example is illustrated in Figure 11 which shows a structured process with two parallel OR-split and OR-joins within a loop. We use that example to show that our OR-join semantics also works with AND-joins since we handle them like OR-joins as mentioned at the beginning of Section 5. For this reason, the AND-join J post-dominates both OR-joins OJ_1 and OJ_2, i.e., both do not wait for one another with regard to our release function. Therefore, the process will be executed as intended.

7 Conclusion

In this paper, we have presented a new definition of soundness for workflow graphs with arbitrary OR-join semantics. Based on this definition, we introduced a quality criterion — completeness — to evaluate semantics of research, especially the semantics of Völzer [2]. Afterwards, a new approach was determined which was proven to be complete. We have shown, that it is (to the best

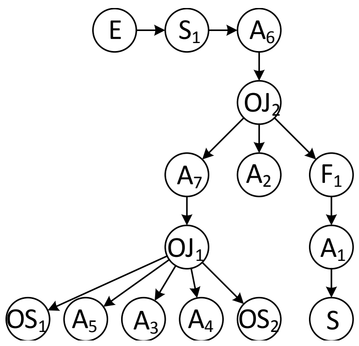


Figure 10. The post-dominator tree of the process of Figure 9

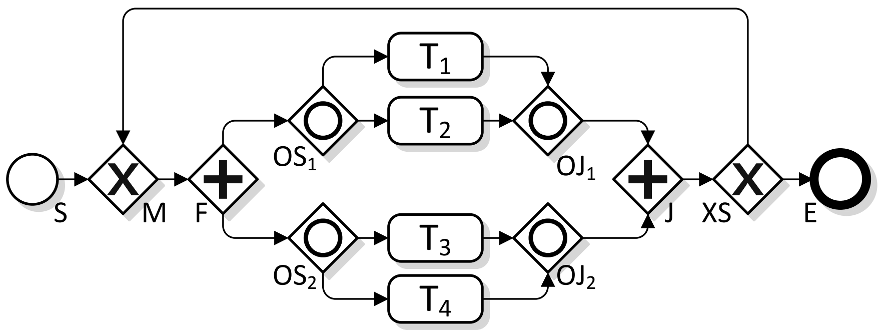


Figure 11. A structured process with an AND-join

author's knowledge) the first, most liberal OR-join semantics which holds for all sound workflow graphs. At the end, it could be shown that two or more OR-joins can actually cause a deadlock in unsound workflow graphs when no correct execution order exists. Our proposed semantics helps to detect such failure situations by static analysis.

Our next step in future work will be to develop a first approach for detecting all deadlocks and states with lack of synchronization within processes containing OR-joins. Hence, we have to extend previous work, e.g., Vanhatalo et al. [14], [16], Fahland et al. [17], and our previous work for process analyses and execution [13], [18], [19], [20].

References

[1] OMG, "Business Process Model and Notation 2.0," *formal/2011-01-03*, 2011. [Online]. Available: http://www.omg.org/spec/BPMN/2.0

[2] H. Völzer, "A new semantics for the inclusive converging gateway in safe processes," in *Business Process Management - 8th International Conference, BPM 2010, Hoboken, NJ, USA, September 13-16, 2010. Proceedings*, ser. Lecture Notes in Computer Science, R. Hull, J. Mendling, and S. Tai, Eds., vol. 6336. Springer, 2010, pp. 294–309. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15618-2_21

[3] E. Kindler, "On the semantics of EPCs: A framework for resolving the vicious circle," in *Business Process Management: Second International Conference, BPM 2004, Potsdam, Germany, June 17-18, 2004. Proceedings*, ser. Lecture Notes in Computer Science, J. Desel, B. Pernici, and M. Weske, Eds., vol. 3080. Springer, 2004, pp. 82–97. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-25970-1_6

[4] E. Kindler, "On the semantics of EPCs: Resolving the vicious circle," *Data Knowl. Eng.*, vol. 56, no. 1, pp. 23–40, 2006. [Online]. Available: http://dx.doi.org/10.1016/j.datak.2005.02.005

[5] J. Mendling and W. M. P. van der Aalst, "Formalization and verification of EPCs with OR-joins based on state and context," in *Advanced Information Systems Engineering, 19th International Conference, CAiSE 2007, Trondheim, Norway, June 11-15, 2007, Proceedings*, ser. Lecture Notes in Computer Science, J. Krogstie, A. L. Opdahl, and G. Sindre, Eds., vol. 4495. Springer, 2007, pp. 439–453. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-72988-4_31

[6] M. Dumas, A. Großkopf, T. Hettel, and M. T. Wynn, "Semantics of standard process models with OR-joins," in *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I*, ser. Lecture Notes in Computer Science, R. Meersman and Z. Tari, Eds., vol. 4803. Springer, 2007, pp. 41–58. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-76848-7_5

[7] N. Cuntz and E. Kindler, "On the semantics of epcs: Efficient calculation and simulation," in *Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings*, W. M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, Eds., vol. 3649, 2005, pp. 398–403. [Online]. Available: http://dx.doi.org/10.1007/11538394_30

[8] M. T. Wynn, D. Edmond, W. M. P. van der Aalst, and A. H. M. ter Hofstede, "Achieving a general, formal and decidable approach to the OR-join in workflow using reset nets," in *ICATPN*, ser. Lecture Notes in Computer Science, G. Ciardo and P. Darondeau, Eds., vol. 3536. Springer, 2005, pp. 423–443.

[9] D. R. Christiansen, M. Carbone, and T. T. Hildebrandt, "Formal semantics and implementation of BPMN 2.0 inclusive gateways," in *WS-FM*, ser. Lecture Notes in Computer Science, M. Bravetti and T. Bultan, Eds., vol. 6551. Springer, 2010, pp. 146–160. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19589-1

[10] W. M. P. van der Aalst and A. H. M. ter Hofstede, "Yawl: yet another workflow language," *Inf. Syst.*, vol. 30, no. 4, pp. 245–275, 2005.

[11] W. Sadiq and M. E. Orlowska, "Analyzing process models using graph reduction techniques," *Inf. Syst.*, vol. 25, no. 2, pp. 117–134, 2000. [Online]. Available: http://dx.doi.org/10.1016/S0306-4379(00)00012-0

[12] W. M. P. van der Aalst, A. Hirnschall, and H. M. W. E. Verbeek, "An alternative way to analyze workflow graphs," in *Advanced Information Systems Engineering, 14th International Conference, CAiSE 2002, Toronto, Canada, May 27-31, 2002, Proceedings*, ser. Lecture Notes in Computer Science, A. B. Pidduck, J. Mylopoulos, C. C. Woo, and M. T. Özsu, Eds., vol. 2348. Springer, 2002, pp. 535–552. [Online]. Available: http://link.springer.de/link/service/series/0558/bibs/2348/23480535.htm

[13] T. M. Prinz and W. Amme, "Practical compiler-based user support during the development of business processes," in *Service-Oriented Computing - ICSOC 2013 Workshops - CCSA, CSB, PASCEB, SWESE, WESOA, and PhD Symposium, Berlin, Germany, December 2-5, 2013. Revised Selected Papers*, ser. Lecture Notes in Computer Science, A. Lomuscio, S. Nepal, F. Patrizi, B. Benatallah, and I. Brandic, Eds., vol. 8377. Springer, 2013, pp. 40–53. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-06859-6_5

[14] J. Vanhatalo, H. Völzer, and F. Leymann, "Faster and more focused control-flow analysis for business process models through SESE decomposition," in *Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17-20, 2007, Proceedings*, ser. Lecture Notes in Computer Science, B. J. Krämer, K. Lin, and P. Narasimhan, Eds., vol. 4749. Springer, 2007, pp. 43–55. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74974-5_4

[15] W. M. P. van der Aalst, "Verification of workflow nets," in *Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN '97, Toulouse, France, June 23-27, 1997, Proceedings*, ser. Lecture Notes in Computer Science, P. Azéma and G. Balbo, Eds., vol. 1248. Springer, 1997, pp. 407–426. [Online]. Available: http://dx.doi.org/10.1007/3-540-63139-9_48

[16] J. Vanhatalo, H. Völzer, and J. Koehler, "The refined process structure tree," in *Business Process Management, 6th International Conference, BPM 2008, Milan, Italy, September 2-4, 2008. Proceedings*, ser. Lecture Notes in Computer Science, M. Dumas, M. Reichert, and M. Shan, Eds., vol. 5240. Springer, 2008, pp. 100–115. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85758-7_10

[17] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf, "Analysis on demand: Instantaneous soundness checking of industrial business process models," *Data Knowl. Eng.*, vol. 70, no. 5, pp. 448–466, May 2011. [Online]. Available: http://dx.doi.org/10.1016/j.datak.2011.01.004

[18] T. M. Prinz, N. Spieß, and W. Amme, “A first step towards a compiler for business processes,” in *Compiler Construction - 23rd International Conference, CC 2014, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Cohen, Ed., vol. 8409. Springer, 2014, pp. 238–243. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54807-9_14

[19] T. M. Prinz, “Proposals for a virtual machine for business processes,” in *Proceedings of the 7th Central European Workshop on Services and their Composition, ZEUS 2015, Jena, Germany, February 19-20, 2015.*, ser. CEUR Workshop Proceedings, T. S. Heinze and T. M. Prinz, Eds., vol. 1360. CEUR-WS.org, 2015, pp. 10–17. [Online]. Available: http://ceur-ws.org/Vol-1360/paper2.pdf

[20] T. M. Prinz, T. S. Heinze, W. Amme, J. Kretzschmar, and C. Beckstein, “Towards a compiler for business processes - a research agenda,” in *SERVICE COMPUTATION 2015: The Seventh International Conferences on Advanced Service Computing*, M. de Barros and C.-P. Rückemann, Eds., vol. 6, IARIA Conference. Nice, France: ThinkMind Digital Library, March 22 2015, pp. 49–54.