Determination of a Correct Execution Semantics for Inclusive Converging Gateways

Thomas M. Prinz and Wolfram Amme
Determination of a Correct Execution Semantics for Inclusive Converging Gateways

Thomas M. Prinz and Wolfram Amme

Friedrich Schiller University Jena, 07743 Jena, Germany
{Thomas.Prinz,Wolfram.Amme}@uni-jena.de

Abstract. Inclusive converging gateways (also known as OR-joins) are frequently used in business processes although their execution semantics still seem unclear. In this paper, we introduce a new execution semantics which is based on an activation and a waiting condition. This model allows a more straightforward definition of state transitions and a precise analysis when a node can be executed. We use it to derive a correct and complete semantics for OR-joins based on the post-dominator relation of the compiler theory. The proof guarantees a correct execution order in structurally correct workflow graphs. Furthermore, our execution semantics allows to compute whether an OR-join can be executed or not in constant time. In conclusion, we show that OR-joins can cause deadlocks.

1 Introduction

One of the major problems in defining a correct execution semantics for business process modeling languages, e.g., Business Process Model and Notation (BPMN) [1] or Event-driven Process Chains (EPCs), is the semantics of inclusive converging gateways, mostly called OR-joins, which are frequently used in business processes [2]. Since the automatic and correct execution of such business processes is impossible without this semantics and since such an automatic execution becomes more and more necessary, finding an universally valid semantics is very important.

The major task of OR-joins in practice is the synchronization of an arbitrary set of control flows. Figure 1 illustrates two business processes in BPMN notation containing OR-joins. The left-hand process (a) has an inclusive diverging gateway $OS_1$ (an OR-split) making it possible to enable the execution of an individual non-empty subset of its successor tasks $T_1$, $T_2$, and $T_3$, where the OR-join $OJ_1$ then combines these flows into a single one. The semantics of the OR-join is intuitively clear since it only has to wait for the arrival of all incoming control flows. This simple example shows that an OR-join has a global 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 [1]. This informal description has two fundamental problems [2]:

1. The state space is used to define whether a control flow may still arrive at a node or not. Since the formalization is based on a complex fix-point theory [3,4], and the exploration of all possible future states has an exponential
worst-case time complexity, the state space approach shows its weakness. The major task is to define the semantics of an OR-join without using the state space.

2. 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].

A more challenging business process is illustrated on the right-hand side \((b)\) of Fig. 1 addressing the introduced problems. It contains a diverging parallel gateway, i.e., a fork, \(F_1\) supplying both OR-joins \(OJ_1\) and \(OJ_2\). The simple execution semantics of the previous example does not hold any more since the OR-join \(OJ_1\) would wait for the execution of \(OJ_2\) and vice versa — a vicious circle. In most popular approaches for execution semantics of inclusive converging gateways [2,5], such a state is interpreted as a *deadlock* since the execution stops. Other approaches declare, that either \(OJ_1\), \(OJ_2\), or both OR-joins should be executed non-deterministically [6].

One may criticize, that vicious circles are not relevant in practice, however, Völzer [2] has shown that such circles actually are used in well-structured business processes (compare Fig. 2). Although the behavior of the business process is intuitive, both OR-joins \(OJ_1\) and \(OJ_2\) mutually depend on each other.

In this paper, we will address all the introduced problems in *structurally correct* business processes. Structural correctness promises the absence of deadlocks in which the execution of a business process is blocked partly and permanently, and the lack of synchronization, i.e., situations in which the same node can be executed twice unintentionally [7,8]. For this, we will introduce a new formalization of an execution semantics which allows a new and straight-forward definition of states and state transitions. It will guarantee the correct decision whether a node can be executed or not in *constant time*.

Based on our formalization of an execution semantics, we will derive a *correct* and *complete* execution semantics for OR-joins in cyclic structurally correct workflow graphs being a more formal representation of business processes. For this, a traditional compiler-based theory — the post-dominator relation — will be used. The proofs show that our proposed execution semantics is the only existing valid one for such workflow graphs. As a consequence, the business process on the right-hand side \((b)\) of Fig. 1 has a correct behavior. Furthermore, the introduced

![Fig. 1](image-url)
execution semantics will show that deadlocks are not a phenomenon of parallel converging gateways (AND-joins) only.

The rest of the paper is structured as follows: Current approaches of executing OR-joins are considered in Section 2. Section 3 summarizes some definitions and notions, followed by the introduction of our new formalization of execution semantics in Section 4 and its application in practice in Section 5. Afterwards, the concrete execution semantics of all node types are derived with the help of compiler-based techniques (Sect. 6). In conclusion, Section 7 retrospects the content of this paper and concludes with a short outlook into future work.

2 Current Approaches

Most business process engines do not support OR-joins without syntactic or semantic restrictions. For example, YAWL [9] allows OR-joins only in structured business processes. In BPEL [10], OR-joins can be used only in the context of acyclic business processes. However, in EPCs and BPMN, OR-joins can be used without restrictions and therefore need a formal semantics. Van der Aalst et al. [11] argue that each formal semantics of OR-joins conflicts with its informal semantics. Kindler [3,4] proposes a state-space interpretation of the OR-join semantics, i.e., if there is a reachable state in the future in which an OR-join gets another control flow, then this OR-join has to wait for this control flow. However, to explore the state-space of a business process, the semantics of an OR-join has to be used. This self-reference was resolved by the fix-point theory. Although this approach seems to be the most general way to define an OR-join semantics, Völzer [2] argues that its definition remains unclear. Furthermore, Cuntz and Kindler [12] state in a subsequent paper that this approach is very inefficient and inapplicable in practice. We will show that a fast-computable OR-joins semantics is absolutely possible.

Other approaches, e.g., the approach of Langner et al. [13] or of Mendling and van der Aalst [5], restrict the business processes to be well-structured. Since well-structured business processes are rare in practice, these approaches seem to be inapplicable. Therefore, we will show that a correct semantics of OR-joins is also possible for unstructured business processes.

Dumas et al. [6] provide a semantics for unstructured business processes which allows to decide in linear time whether an OR-join is allowed to be executed.

Fig. 2. A vicious circle in a well-structured business process [2]
or not. The approach is based on a graph-based interpretation of the informal semantics "may arrive". It includes all direct and indirect predecessors of an OR-join in such that an OR-join is executable when neither its direct nor its indirect predecessors have a control flow. In conflict situations, i.e., vicious circles, in which two OR-joins mutually wait for each other, this conflict will be resolved non-deterministically. We will show that such a conflict avoidance can result in a lack of synchronization, i.e., the intuitive semantics of these OR-joins remains unclear. Since such conflicts can happen in well-structured business processes as mentioned before, the approach is inapplicable as well.

Völzer [2] uses a similar graph-based approach, however, the business process is separated into blocks. Therefore, an OR-join closing such a block has only to wait for control flows being inside the block. In non-separable process parts, two or more OR-joins may have a deadlock if they are mutually dependant on each other. The author argues, that the intuitive meaning of this business process is unclear. As mentioned before, we will show that the decision whether an OR-join can be executed or not is independent from such block structures, and that this approach only provides an estimation. Furthermore, the semantics defined by Völzer is a special case of our proposed semantics.

The basic assumption of the approach of Völzer is, that a semantics of OR-joins should avoid structural conflicts, i.e., deadlocks and lack of synchronization. We adhere to this assumption since OR-joins should be usable as merges and AND-joins, and therefore should not cause lack of synchronization or deadlocks in general. Furthermore, our approach is also graph-based and works with unstructured and cyclic business processes.

3 Preliminaries

In this section, we introduce some basics of compiler theory and business process verification. Business processes are mostly represented as workflow nets [14] being a special kind of Petri nets, or workflow graphs, which are special cases of control flow graphs [7] allowing explicit parallelism. We prefer workflow graphs since the modeling of inclusive gateways in Petri nets is difficult [15].

A workflow graph is a directed graph $WFG = (N, E)$ such that $N$ consists of activities, forks, AND-joins, splits, merges, OR-splits, OR-joins, one start and one end node. The end node, activity, split, OR-split, and fork each have exactly one incoming edge; whereas the start node, activity, merge, OR-join, and AND-join each have exactly one outgoing edge. Splits, OR-splits, and forks have at least two outgoing edges; and merges, OR-joins, and AND-joins have at least two incoming edges. Furthermore, each node lies on a path from the start to the end node.

![Fig. 3. A workflow graph](image)
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. In conclusion, OR-splits and OR-joins are depicted as (thick) diamonds with black dots.

The semantics of workflow graphs used in this paper is similar to the semantics of control flow graphs: the execution of a workflow graph starts at the start node and follows the flow described by the directed graph. An activity, a split, a merge, a fork, and the end node can be executed if a control flow reaches one of their incoming edges, whereas an AND-join can only fire if all of its incoming edges are reached by a control flow. In workflow graphs that consider no data-aspects, the split decides non-deterministically which outgoing edge is followed by the control flow. After the execution of a fork, parallel control flows are created for each outgoing edge. The semantics of an OR-split is similar to the semantics of splits deciding non-deterministically which non-empty subset of outgoing edges is followed by the control flow. Since the semantics of OR-joins are the subject of this report, at this moment, it is only assumed that an OR-join can synchronize a subset of control flows and waits for those control flows which may reach it.

Paths will be used to describe control flows within workflow graphs. Formally, a path $P = (n_1, n_2, \ldots, n_{m-1}, n_m)$ is a sequence of pairwise different nodes of $N$ such that $\forall i \in \{1, \ldots, m-1\}: (n_i, n_{i+1}) \in E$. We write $n \in P$ if $n \in \{n_1, \ldots, n_m\}$.

3.1 Structural Correctness

Structural correctness (soundness) is currently the most important correctness criterion of business processes. It describes the absence of deadlocks, i.e., situations in which the execution within business processes is partly blocked, and lack of synchronization in which parts of business processes are executed twice unintentionally [7]. Deadlocks and lack of synchronization prevent the proper termination of business processes — deadlocks block control flows, and in a lack of synchronization, at least two control flows reach the same node of business processes while it is unclear which control flow should be followed. A workflow graph is structurally correct if it neither has deadlocks nor lack of synchronization.

In previous work [8, 16], we have shown a detailed and fast failure analysis for workflow graphs without OR-joins by searching potential deadlocks and lack of synchronization, which can be structurally identified. Structurally correct workflow graphs have the advantage that each parallelism is synchronized by an AND-join or an OR-join, and no deadlock can prevent their proper execution.
3.2 Compiler Theory Basics

In compiler theory, the definition of the post-dominance relation is the following:

Definition 1 (Post-Dominance). Let $WFG = (N, E)$ be a workflow graph with its start node $s$ and its end node $e$.

A node $n$ post-dominates node $m$ if every path from $m$ to $e$ contains $n$, written $n \text{ pdom } m$. $n$ properly post-dominates $m$ if $n \text{ pdom } m$ 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 $\text{pdom}(WFG) = (N, E')$ with $E' = \{(n, m) : n$ directly post-dominates $m$ in $WFG\}$.

The set $\text{pdom}(n) = \{m : m \text{ pdom } n\}$ can be computed in nearly linear time using a depth-first search [17].

4 Execution Semantics

As mentioned before, we introduce a new formalization of an execution semantics. Therefore, an analysis of the conditions under which a node can be executed or not is needed. The decision whether an arbitrary node can be executed or not is usually divided into two parts: The first part checks whether a control flow has reached an incoming edge of this node, i.e., the node is activated to be executed in future. That is why we call it the activation condition. In the second step, it has to be checked whether this node has not to wait for other nodes, e.g., the execution of its predecessor nodes if it is an AND-join. Therefore, we call it the waiting condition. As a consequence, each arbitrary node is executable if the activation condition is fulfilled and the waiting condition fails. The following definition describes the formal model of our execution semantics in structurally correct workflow graphs.

Definition 2 (Execution Semantics). Let $WFG = (N, E)$ be a structurally correct workflow graph.

An execution semantics of $WFG$ is a pair, $\mathcal{S} = (\alpha\text{-cond}, \omega\text{-cond})$, such that

$\alpha\text{-cond}$ is the activation condition of each node, and
$\omega\text{-cond}$ is the waiting condition of each node.

A node $n$ within $WFG$ is allowed to be executed iff the following formula holds

$$\alpha\text{-cond}(n) \land \neg \omega\text{-cond}(n)$$

In structurally incorrect workflow graphs, we need an additional condition to avoid the execution of an AND-join to happen too early, i.e., that control flows do not arrive on all of its incoming edges.
In general, when a control flow arrives at one of the incoming edges of an arbitrary node, the node will always be executed in future structurally correct workflow graphs, i.e., it is activated. The formalization of this condition, the activation condition, is as follows:

**Definition 3 (Activation Condition).** Let $WFG = (N, E)$ be a structurally correct workflow graph.

The mapping $\alpha$-cond: $N \rightarrow \{\text{false}, \text{true}\}$ is called activation condition. The activation condition is defined for each node $n$ by

$$\alpha$\text{-cond}(n) = \begin{cases} \text{true}, & n \text{ is the start node and } WFG \text{ was initiated} \\ \text{true}, & \exists e = (x, n) \in E, x \text{ is predecessor of } n: e \text{ has the control} \\ \text{false}, & \text{otherwise} \end{cases}$$

We say that a node $n$ will be active iff $\alpha$-cond$(n) = \text{true}$.

Activities, forks, splits, OR-splits, merges, the start node, and the end node can be executed immediately after their activation. However, AND-joins and OR-joins have to wait for the execution of some previous nodes. Basically, a node has to wait for another node if the control flow of the latter may arrive at the former.

**Definition 4 (Waiting).** Let $WFG = (N, E)$ be a structurally correct workflow graph, and $n_1, n_2$ two of its nodes.

The node $n_2$ waits on $n_1$ iff $n_1$ and $n_2$ are active and the control flow activating $n_1$ may arrive at $n_2$.

A node has to check whether each node it is waiting for is active. The set of these nodes is called its waiting area.

**Definition 5 (Waiting Area).** Let $WFG = (N, E)$ be a structurally correct workflow graph.

The mapping $\omega$ : $N \rightarrow \mathcal{P}(N)$ is called waiting area and defines the set of nodes which an active node $n$ has to observe since $n$ cannot be executed as long as one of them is active.

The waiting condition can be generally defined by the use of waiting areas.

**Definition 6 (Waiting Condition).** Let $WFG = (N, E)$ be a structurally correct workflow graph.

The mapping $\omega$-cond: $N \rightarrow \{\text{false}, \text{true}\}$ is called waiting condition. The waiting condition is defined for each node $n$ by

$$\omega$\text{-cond}(n) = \begin{cases} \text{true}, & \exists m \in \omega(n): m \text{ is active} \\ \text{false}, & \text{otherwise} \end{cases}$$
Our formal representation of an execution semantics may be used for the definition of other execution semantics, e.g., the state space approach. In the state space approach, each node has to wait for control flows which may still arrive. More formally, it has to wait for the execution of all nodes with a path toward it. Therefore, its waiting area consists of all these nodes. However, as mentioned before, this execution semantics would terminate in a deadlock during the execution of the business process of Fig. 2.

In the next sections, we show that this formal representation of an execution semantics can be implemented efficiently, and that we can construct an implementation of waiting areas for each node defining a correct semantics.

5 Semantics in Practice

In the previous section, a formalization of an execution semantics was introduced which is based on an activation condition $\alpha$-cond and a waiting condition $\omega$-cond. As mentioned before, the activation condition of a node evaluates to true when a control flow reaches one of its incoming edges—more precisely if one of the node’s direct predecessors was executed and the predecessors control flow arrives at it. Then, the set of all active nodes within a workflow graph is a representation of the state of its execution.

**Definition 7 (State).** Let $WFG = (N, E)$ be a structurally correct workflow graph, $s$ its start, and $e$ its end node.

A state of $WFG$ is a subset of $N$ in which each node is active. The initial state is defined as the set $\{s\}$, the termination state is the set $\{e\}$.

Each execution of a node activates some of its direct successor nodes, whereby a split and OR-split decides non-deterministically which successor nodes will be activated. Therefore, we can define a state change as follows:

**Definition 8 (State Changes).** Let $WFG = (N, E)$ be a structurally correct workflow graph.

A state change from state to state’ after executing the node $n$, depicted as state $\xrightarrow{n}$ state’, is the execution of the following formula in which $\text{succ}^*$ represents a subset of $n$’s direct successor nodes:

$$state' = state \setminus \{n\} \cup \text{succ}^*$$

We write state $\xrightarrow{\ast}$ state’ iff state’ is reachable from state and there is a (possibly empty) finite sequence state $\xrightarrow{n_1} S_1 \ldots S_{k-1} n_k$ state’.

A state can be efficiently represented as a bit set. Each node of the workflow graph gets an unique ascending number starting from zero. A node with the unique number $num$ is active if the bit set on position $num$ contains the value true. If it is inactive, the value is false. On the left-hand side of Fig. 4 there is an acyclic workflow graph whose state is represented by control flows (filled cycles) on incoming edges. The upper right-hand side shows an implementation.
of the active nodes as a bit set. It states that the nodes $A_1$, $O_1$, and $O_2$ are active corresponding with the state in the workflow graph.

As mentioned above, it is determined by the evaluation of the waiting condition whether an active node can be executed or not in structurally correct workflow graphs. The waiting condition evaluates to false when no node within its waiting area is active. Since the waiting area was introduced as a set of nodes, it can be represented also as a bit set similar to the state of a workflow graph. Then, the waiting area of a node contains another node with the unique number $num$ if the value on position $num$ is true, and not if the value is false. In other words, each node has a bit mask representing its waiting area. The right-hand side of the example in Fig. 4 shows the waiting area of the OR-join $O_1$, which states that it has to wait for the nodes $S$, $F_1$, $A_1$, and $A_2$. The details of how the waiting area is defined will be derived in the next section.

However, these considerations can be used to define a fast algorithm which executes a workflow graph such that each decision whether a node can be executed or not is performed in constant time. Such a decision is based on the fact that a node can be executed if it is active and does not have to wait. Since the state of a workflow graph is represented as a set (bit set), the activation condition can be checked with a simple lookup on this state, i.e., a node $n$ is active if $n \in state$. Furthermore, the waiting condition of a node $n$ can be evaluated swiftly with an intersection operation of the state and its waiting area. If the intersection is empty, this node can be executed; more formally if $\omega(n) \cap state = \emptyset$. The right-hand side of Fig. 4 shows the computation of the intersection set for OR-join $O_1$, which has to wait for $A_1$.

The activation and the waiting condition are computable in constant time which makes the definition of the algorithm shown in Fig. 5 possible.
Input: A workflow graph $WFG$ with start node $start$ and end node $end$
Output: An execution of $WFG$

1: $state \leftarrow \{start\}$  
2: while $\exists node \in state \setminus \{end\}$ do  
3: \hspace{1em} if $\omega(node) \cap state = \emptyset$ then  
4: \hspace{2em} execute node  
5: \hspace{1em} if $node \in N_{splits}$ then  
6: \hspace{2em} succ is a non-deterministically chosen successor of $node$  
7: \hspace{2em} $state \leftarrow state \cup \{succ\}$  
8: \hspace{1em} else if $node \in N_{OR-splits}$ then  
9: \hspace{2em} succ* is a non-deterministically chosen subset of successors of $node$  
10: \hspace{2em} $state \leftarrow state \cup succ*$  
11: \hspace{1em} else  
12: \hspace{2em} $state \leftarrow state \cup \{succ: succ \text{ is successor of } node\}$  
13: \hspace{1em} end if  
14: \hspace{1em} $state \leftarrow state \setminus \{node\}$  
15: end if

Fig. 5. Execution of a workflow graph

6 Waiting Areas

In the remainder of this section, we use the term 'post-dominate' as a synonym for 'properly post-dominate' and redefine the set $pdom(n)$ similarly. Furthermore, we handle AND-joins as a special case of OR-joins for which control flow is always reached on all incoming edges. This is possible in structurally correct workflow graphs and therefore we refer to all OR- and AND-joins as $OR-joins$. Finally, we adhere to the assumption that there is a structurally correct workflow graph $WFG$.

In the previous sections, we have defined our execution semantics with an activation and a waiting condition belonging to the waiting area of a node. The waiting area of a node $n$ is a set of nodes which $n$ has to observe since it cannot be executed as long as one of these nodes is active. In this section, we define the waiting areas of each kind of node.

For the determination of the waiting area of a node $n$, we use an estimation, i.e., that we construct the waiting area starting from the set of all nodes of a workflow graph by successively eliminating those nodes whose control flow cannot arrive at $n$. Actually, such a constructed waiting area consists of some nodes which are never active at the same moment as $n$. However, this results never in an abnormal behavior since if a node is not active at the same moment as $n$, $n$ does not finally wait for it.

In the following, we introduce some basic properties when a node, especially an OR-join, must not wait for another node. As a consequence of these properties, there are situations in which an OR-join must not wait for another OR-join as it would result in a deadlock. With this in mind, we subsequently define an appropriate execution order for such situations.
6.1 Basic Properties

Since the arrival of an additional control flow at an active node would cause a lack of synchronization (as long as it is not an OR-join), the waiting area of these kinds of nodes is the empty set. Furthermore, a node must never wait for another node if there is no path in-between as no control flow ever arrives at it.

Remark 1. Let $n_1, n_2, n_1 \neq n_2$, be two nodes.

\[(a)\] $n_1$ is not an OR-join $\implies n_1$ must never wait for another node

\[(b)\] There is no path from $n_1$ to $n_2$ $\implies n_2$ must never wait for $n_1$

As a result from the previous remark, an active node can be executed (as long as it is not an OR-join) and it activates at least one arbitrary direct successor node.

Remark 2. Let $n$ be a node which is active in state $S$ and not an OR-join, and $\text{succ}$ one of $n$'s direct successor nodes.

$n$ will be executed $\implies S \xrightarrow{n} S', \text{succ} \in S', S \setminus \{n\} \subseteq S'$, is possible

Then, a path without OR-joins can be executed as soon as the first node of this path is active.

Corollary 1 (Execution Sequence). Let $P = (n_0, \ldots, n_{k-1}, n_k), k \geq 1$, be a path, and $n_0$ is active in state $S_0$.

None of $n_0, \ldots, n_{k-1}$ is an OR-join $\implies S_0 \xrightarrow{n_0} S_1 \ldots S_{k-1} \xrightarrow{n_{k-1}} S_k$ is possible with $n_k \in S_k$, and $(S_{k-1} \setminus \{n_{k-1}\}) \subseteq S_k$

In consequence of the previous remarks and the corollary, the major task is to define under which conditions an OR-join must not wait for another node. The following lemma describes an interesting property which holds in this respect for OR-joins in structurally correct workflow graphs.

Lemma 1. Let $j$ be an active OR-join and $n$ another active node.

$j$ must not wait for $n$ $\implies$ On all paths from $n$ to $j$, an OR-join having to wait for $j$ can be activated
Proof (Lemma 1). Constructive proof. In general, there are two possibilities regarding Remark 1 that \( j \) must not wait for \( n \):

Possibility 1: There is no path from \( n \) to \( j \). ✓

Possibility 2: There are paths from \( n \) to \( j \). Let \( P_i \) be one of those paths. It follows that the control flow of \( n \) can never arrive at \( j \) via \( P_i \) since either the control flow is blocked, it always leaves \( P_i \), or it activates a node which has to wait for \( j \).

Case 1: The control flow will be blocked permanently, i.e., a deadlock occurs. ✗

Case 2: The control flow always leaves \( P_i \). There must be a split or an OR-split on \( P_i \) which always decides that the control flow does not follow \( P_i \). Since splits and OR-splits decide non-deterministically, this case is not possible. ✗

Case 3: The control flow activates a node which has to wait for \( j \). This node must be an OR-join \( j_2 \) since it is able to wait. Since there would be a deadlock if \( j \) also waits for \( j_2 \), \( j \) cannot wait for \( j_2 \). Therefore, it must not wait for the control flow of \( n \) via \( P_i \). ✓

□

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

**Lemma 2.** Let \( j \) be an active OR-join and \( n \) another active node.

On all paths from \( n \) to \( j \), an OR-join having to wait for \( j \) can be activated

\[ \implies \]

\( j \) must not wait for \( n \)

**Proof (Lemma 2).** Constructive proof.

\[
\begin{array}{c}
  \text{Case 1: There is no path from } n \text{ to } j. \\
  \text{Def. } 4 \implies j \text{ must not wait for } n. \checkmark
  \end{array}
\]

\[
\begin{array}{c}
  \text{Case 2: There are paths from } n \text{ to } j \text{ (compare with the previous figure)} \implies \\
  \text{if the control flow of } n \text{ followed path } P_i \text{ to } j, \text{ it would} \\
  \text{activate an OR-join } w_i \text{ which would have to wait for } j \\
  \text{deadlock-free } \implies j \text{ must not wait for } w_i \\
  \implies j \text{ can be executed before } w_i \\
  \implies j \text{ must not wait for } n. \checkmark
  \end{array}
\]

□
The following theorem summarizes the last two lemmas:

**Theorem 1.** Let $j$ be an active OR-join and $n$ another active node.

$j$ must not wait for $n$  

$\iff$

On all paths from $n$ to $j$, an OR-join having to wait for $j$ can be activated

*Proof (Theorem 1).* Follows from Lemma 1 and 2.

---

### 6.2 Determination of an Appropriate Execution Order

As already mentioned, in contrast to the waiting areas of nodes like activities and merges, the definition of the waiting area of an OR-join is still open. In the acyclic case, the definition of the waiting area of an OR-join $j$ is simple since it contains all nodes, which have a path to $j$, as there is no deadlock blocking the arrival of the control flow. In contrast, in the cyclic case, two OR-joins may mutually wait for each other and therefore cause a deadlock. These situations are called *vicious circles* [3]. For the avoidance of such circles, the definition of a correct execution semantics have in principle to describe an appropriate execution order deciding which OR-join has to wait and which OR-join must not wait. The next theorem describes an important property in which an OR-join must wait for other active nodes which it post-dominates.

**Theorem 2.** Let $j$ be an active OR-join and $n$ another active node.

$j$ lies on all paths from $n$ to the end node $(j \text{ pdom } n)$  

$\implies$

$j$ must wait for $n$

*Proof (Theorem 2).* Let $\text{pdom}(WFG)$ be the post-dominator tree of $WFG$. $\text{pdom}(WFG)$ defines a partial order caused by the direct post-dominance relation.

We use this order to assign numbers to each OR-join: Each OR-join $j$ is given the number of OR-joins lying on the path from $j$ to the root node. For example, an OR-join labelled $d$ has $d$ predecessors being OR-joins in $\text{pdom}(WFG)$. We write $j_d$ to describe an OR-join $j$ with $d$ predecessors being OR-joins in $\text{pdom}(WFG)$. We can reformulate the hypothesis of this lemma as follows if $j_d$ is an OR-join and $j_d$ and $n$, $j_d \neq n$, are active.

$j_d \text{ pdom } n$  

$\implies$

$j_d$ must wait 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 \( n \)
- \( j_0 \) can be executed independently from \( n \)
- \( j_0 \) can be executed before \( n \)

Assume that \( j_0 \) is executed before \( n \)

Cor. \( 1 \) ⇒ the control flow activating \( j_0 \) can activate the \( end \) node since there is a path \( P \) from \( j_0 \) to \( end \) without another OR-join
- \( end \) and \( n \) could be active
- lack of synchronization since the control flow activating \( n \) and the control flow activating \( end \) will never be synchronized

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 following figure reflects the overall situation:

Assumption: \( j_{d+1} \) does not have to wait for \( n \)
- \( j_{d+1} \) can be executed independently from \( n \)
  and again can be executed before \( n \)

Assume that \( j_{d+1} \) is executed before \( n \)

Cor. \( 1 \) ⇒ the control flow activating \( j_{d+1} \) can activate \( j_d \) since there is a path \( P \) from \( j_{d+1} \) to \( j_d \) without an OR-join
- hypothesis ⇒ \( j_d \) must wait for \( n \)
- deadlock-free ⇒ \( n \) must not wait for \( j_d \)
  ⇒ the control flow of \( n \) can arrive at \( j_{d+1} \)
  ⇒ the same path \( P \) from \( j_{d+1} \) to \( j_d \) without another OR-join can be followed once more by the control flow
  ⇒ there can be a lack of synchronization at \( j_d \)

As result, the following corollary can be derived directly from the previous Theorem \( 2 \) since two OR-joins mutually waiting for each other cause a deadlock.

**Corollary 2.** *Let \( j \) be an active OR-join and \( n \) another active node.*

\[ n \text{ is post-dominated by } j \]

⇒

\[ n \text{ must not wait for } j \]
In principle, Corollary 2 describes that in a vicious circle an OR-join does not have to wait for another OR-join if it is post-dominated by this node. For the moment, we assume that vicious circles have only an appropriate execution order if one OR-join post-dominates the other. The correctness of this assumption will be proven afterwards in Theorem 3. Under these terms, the following definition results, in particular, from the previous corollary describes the waiting area of an OR-join.

**Definition 9 (Waiting Area of OR-Joins).** Let \( j \) be an OR-join.

The waiting area of \( j \) is the set \( \omega(j) = \{ n : \text{there is a non-empty path from } n \text{ to } j \text{ and on this path lies no OR-join which post-dominates } j \} \).

As a result from Theorem 1, this definition implicitly makes use of the fact that an active OR-join \( j \) does not have to wait for an active node \( n \) if there is an OR-join post-dominating \( j \) on all paths from \( n \) to that \( j \).

Figure 6 shows a workflow graph with two OR-joins \( O_1 \) and \( O_2 \) within a vicious circle which can be resolved with our execution semantics. Since the OR-join \( O_2 \) post-dominates the OR-join \( O_1 \), \( O_2 \) must wait for \( O_1 \). Therefore, the waiting areas of both nodes are \( \omega(O_1) = \{ S, F_1, A_1, A_3, S_1 \} \) and \( \omega(O_2) = \{ S, F_1, A_1, A_2, A_3, A_5, S_1 \} \) by Def. 9. In this case, this execution order will not result in a lack of synchronization or deadlock and therefore leads to a structurally correct workflow graph.

As argued in Section 3, the identification of all post-dominators for each node can be computed in nearly linear time. Furthermore, for each OR-join an inverse depth-first search has to be made which stops its traversal at the start node and before finding an OR-join post-dominator. Therefore, if the size of the set of edges is linear to the set of nodes as in most business processes, the computation of each waiting area can be completed in nearly linear time for a single node. In addition, the computation of waiting areas have to be completed once before an execution engine starts the execution of the workflow graph. In summary, our execution semantics is very efficient.

When defining the waiting areas of OR-joins, our assumption was that vicious circles can only be resolved if one of the OR-joins post-dominates the other. In the following theorem, it will be proven that this assumption is correct, i.e., there is no appropriate execution order, otherwise.
Theorem 3. Let $j_1, j_2, j_1 \neq j_2$, be two active OR-joins which have at least one path to each other.

$j_1$ does not post-dominate $j_2$ and $j_2$ does not post-dominate $j_1$ 

$\implies$ 

$WFG$ is structurally incorrect

Proof (Theorem 3). Constructive proof.

Case a: $j_1$ has to wait for $j_2$ and $j_2$ has to wait for $j_1$.
This is a deadlock since the execution is blocked permanently. ✓

Case b: $j_1$ has not to wait for $j_2$ or $j_2$ has not to wait for $j_1$
Without loss of generality, we assume that $j_2$ has not wait for $j_1$, i.e., $j_2$
is executed before or at the same time as $j_1$. Furthermore, there is a path $P'$ from $j_2$ to the end node without $j_1$ since $j_1$ does not post-dominate $j_2$. There is also a path $P$ from $j_1$ to the end node which contains $P'$ as a subpath since there is a path from $j_1$ to $j_2$. The following figure illustrates the situation conceptually.

Since $j_2$ has not to wait for $j_1$, let $j_2$ be executed before or at the same time as $j_1$. As a consequence, in both cases, $j_2$ is executed and one of its direct successor nodes is activated. A possible situation 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.

We assume a variable $x$, which describes the current node being activated by the control flow of $j_1$ on path $P$, and a variable $y$ which describes the current node being activated by the control flow of $j_2$ on the same path $P$. At the beginning, $x$ is $j_1$ or the direct successor node of $j_1$ and $y$ is the direct successor node of $j_2$.

In each execution step of variables $x$ and $y$, there are two possibilities:
Case 1: $x$ has to wait for $y$ and $y$ has to wait for $x$.

$\implies$ deadlock since the execution is blocked partly

Case 2: $x$ has not to wait for $y$ or $y$ has not to wait for $x$.

$\implies$ $x$ or $y$ will be executed (possibly at the same time)

$\implies$ after executing $x$ or $y$, the direct successor node of $x$ or $y$ which lies on path $P$ can be activated
Both cases show that there can be either a deadlock or both control flows can exclusively work on possible different nodes of path $P$. In the case of a deadlock, $WFG$ cannot be structurally correct. Otherwise, this would lead to a lack of synchronization since both control flows must eventually arrive at the same incoming edge of a node of path $P'$. This results also in a structurally incorrect workflow graph.

In all cases, an execution may result in a structural error which is not possible in structurally correct workflow graphs. □

Fig. 7 shows a workflow graph with a vicious circle $O_1 \rightarrow S_2 \rightarrow O_2 \rightarrow S_1 \rightarrow O_1$ and demonstrates that deadlocks are not exclusively a phenomenon of AND-joins. At first glance, this workflow graph seems to be structurally correct since there is no AND-join which can cause a deadlock, and on all paths to the end node, the control flows produced by the fork $F_1$ seem to be synchronized. Since neither $O_1$ post-dominates $O_2$ nor $O_2$ post-dominates $O_1$, the waiting areas of those OR-joins are $\omega(O_1) = \{S,F_1,A_1,A_2,O_2,S_2\}$ and $\omega(O_2) = \{S,F_1,A_1,A_2,O_1,S_1\}$. 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. This situation causes a deadlock since the execution of the workflow graph is blocked. As stated in Theorem 3, the workflow graph cannot be structurally correct. For example, if we allow one or both of the OR-joins to be executed, the two control flows may arrive at the same incoming edge of the OR-join $O_3$. In summary, all deadlock analyses have to be extended in order to detect this new kind of structural deadlocks in workflow graphs.

7 Conclusion

In this paper, we have presented a correct and complete semantics for OR-joins in structurally correct workflow graphs. We have introduced a new execution semantics which is based on two conditions, the activation and the waiting condition, and allows to decide in constant time whether a node can be executed or not. Based on this, we derived a correct and complete activation and waiting condition of OR-joins. At the end, it could be shown that two or more OR-joins can actually cause a deadlock when no correct execution order exists. Therefore,
these deadlock situations are a new kind of deadlocks within workflow graphs and have to be considered by static analyses.

Our next step in future work will be to develop a first approach for detecting all deadlocks and lack of synchronization within business processes containing OR-joins. In this sense, we have to extend our previous work [8,16].

References

