RZ 3791 (# Z1011-001) 11/18/2010
Computer Science 20 pages

Research Report

A New Semantics for the Inclusive Converging Gateway in Safe Processes

H. Völzer

IBM Research – Zurich
8803 Rüschlikon
Switzerland

A New Semantics for the Inclusive Converging Gateway in Safe Processes

Hagen Völzer

IBM Research — Zurich, Switzerland
hvo@zurich.ibm.com

Abstract. We propose a new semantics for the inclusive converging gateway (also known as *Or-join*). The new semantics coincides with the intuitive, widely agreed semantics for Or-joins on sound acyclic workflow graphs which is implied, for example, by dead path elimination on BPEL flows. The new semantics also coincides with the block-based semantics as used in BPEL on cyclic graphs that can be composed from sound acyclic graphs, repeat- and while-loops. Furthermore, we display several examples for unstructured workflow graphs for which Or-joins get the desired intuitive semantics. A key insight is that not all situations where two or more Or-joins seem to be mutually dependent (known as 'vicious circles') are necessarily symmetric. Many such situations are asymmetric and can be resolved naturally in favor of one of the Or-joins. Still symmetric or almost symmetric situations exist, for which it is not clear what semantics is desirable and which result in a deadlock in our semantics. We show that enabledness of an Or-join in our semantics can be decided in linear time in the size of the workflow graph.

1 Introduction

The semantics of the inclusive converging gateway, also known as *Or-join*, is recognized as one of the main problems of defining an execution semantics for a business process modeling language that permits unrestricted directed graphs such as BPMN and EPCs. With an increased interest in directly executing BPMN models or generating code from them, the problem has become more important and consequently, has received a lot of attention recently [7–9, 3, 4, 2, 1].

Or-joins were introduced into business process modeling languages to be able to synchronize a variable set of threads. The simplest example is shown in Fig. 1, which

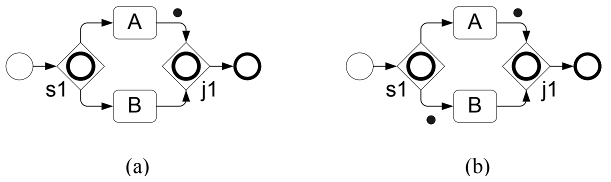


Fig. 1. A simple example of an Or-join

is drawn using BPMN [5]. The Or-split $s1$ produces a token for either or both of its outgoing edges, i.e., either task *A* will be enabled, or task *B* or both of them. The corresponding Or-join $j1$ is meant to synchronize the created threads, i.e., it should wait for all tokens that were created by the Or-split $s1$ before it consumes them and produces a token on its outgoing edge. This simple example already shows that the semantics of the Or-join is necessarily *non-local*, i.e., in contrast to all other usual gateways, its enabledness does not only depend on the tokens on its incoming edges. To see that, consider the states shown in Fig. 1(a) and (b), where a token on an edge is shown as a black dot. The Or-join behaves differently in both states: In (a), $j1$ is enabled whereas in (b), $j1$ is not enabled as it has to wait for the other token that has not yet passed task *B*. However both states are indistinguishable if we only look at the incoming edges of $j1$. Therefore, the semantics of the Or-join is non-local.

Figure 2 shows more examples for using an Or-join to synchronize a variable set of threads. In other words, the Or-join merges a set of paths that are neither necessarily pairwise alternative, in which case an Xor-join can be used, nor are the paths pairwise parallel, in which case an And-join can be used. Figure 2 shows that no Or-split is needed to create such a variable set of paths.

We henceforth do not show any tasks in our examples because they are not relevant for our considerations. The reader may imagine a task on each edge of the graph.

The Or-join is often used without a formalized semantics. The informal statement that is usually given to explain its intended behavior is that the Or-join has to wait for all tokens that *'may still arrive'* on its incoming edges, cf. [7]. This statement raises at least two fundamental problems:

1. Whether a token may still arrive is traditionally interpreted on the state space of the graph [7–9, 3]. Van der Aalst, Desel, and Kindler [7] have shown that a straightforward formalization of the informal statement then fails because 'may still arrive' implicitly refers to the very semantics that it is used to define. The same authors [7] and later, Kindler [3], showed that this self-reference can be resolved using fixed-point theory. While it remains unclear whether the resulting semantics is useful for all graphs, there is still another problem with the state-space interpretation: In order to determine whether an Or-join is enabled in a given state s of the graph, one has to explore in the worst case all possible future states of s, of which there are exponentially many. This would potentially pose a substantial problem for an execution engine.

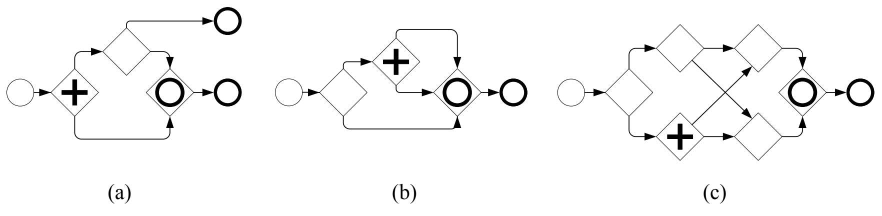


Fig. 2. Or-joins for synchronizing a variable set of threads

2. The second fundamental problem is that two or more Or-joins may mutually depend on each other: One join would be enabled only if the other would not and vice versa. Such situations, which can only occur in cyclic graphs, have been called 'vicious circles' [7]. A typical example is shown later in Fig. 5. It is not clear whether models containing such situations actually occur in practice and if so, what they are intended to model and hence how the semantics should be defined for them.

We address the first problem by using a graph-based interpretation of 'may arrive', which is inspired by informal statements in version 1.0 of the BPMN specification [5]. A similar approach has also been proposed by Dumas *et al.*[2]. This resolves the formalization problem and results in a low computational complexity of enactment. Dumas *et al.*[2] obtain an algorithm that runs in quadratic time which can be reduced to linear time after constructing a data structure of quadratic size. We will display a simple linear time algorithm for our new semantics.

Regarding the second problem, we argue that 'vicious circles' can occur in practice by displaying simple reasonable graphs that contain vicious circles. Even simple *well-structured* graphs, i.e., graphs that are composed from matching pairs of splits and joins, may contain a vicious circle. Existing proposals for Or-join semantics, including Dumas *et al.*[2], then introduce either a deadlock or an artificial non-deterministic choice to such well-structured graphs, which we do not find satisfactory. We argue that a well-structured graph has a natural semantics as it can be seen as a representation of a block-structured process, e.g. modeled in BPEL, and it can be executed accordingly as implied, e.g., by the semantics of BPEL. This was also stated as a design goal for an Or-join semantics by Mendling and van der Aalst [4].

We show how the natural semantics for well-structured graphs can be defined without referring to blocks, which gives rise to a new semantics for general workflow graphs. The new semantics agrees with the widely accepted Or-join semantics for acyclic graphs, which is implied, for example, by the semantics for BPEL flows, i.e., *dead path elimination*. The new semantics also agrees with the above mentioned natural semantics for well-structured graphs. Moreover, we display examples of unstructured cyclic graphs that also get a desired intuitive semantics. Still some models with vicious circles remain, for which it is not clear what a reasonable semantics should be. These cases create a deadlock in our semantics and hence could be sorted out by static analysis.

Our semantics was developed as part of the BPMN 2.0 standardization effort and it is included in the current BPMN 2.0 specification draft [6].

The paper is structured as follows. After introducing preliminary notions in Sect. 2, we discuss Or-join semantics for acyclic graphs in Sect. 3. Acyclic graphs form an important class because essentially, it is intuitively clear how Or-joins should behave in acyclic graphs, viz. as defined by dead path elimination (see Sect. 3.3). As said above, 'vicious circles' do not occur in acyclic graphs. Then, in Sect. 4, we stepwise introduce the new semantics and discuss its various aspects. Some proofs are omitted in the main text but can be found in the appendix.

2 Preliminaries

This section defines the basic preliminary notions of this paper, which include workflow graphs, their semantics and the *soundness* property for workflow graphs. Unsound workflow graphs are usually considered as invalid models that contain modeling errors.

A *workflow graph* $G = (V, E, \ell)$ consists of set V of *nodes*, a set $E \subseteq V \times V$ of *edges*[1], and a partial mapping $\ell : V \to \{\text{And}, \text{Xor}, \text{Or}\}$ such that

1. $\ell(x)$ is defined if and only if x has more than one incoming edge or more than one outgoing edge,
2. there is exactly one source and at least one sink,
3. the source has exactly one outgoing edge and each sink has exactly one incoming edge, and
4. every node is on a path from the source to some sink.

The source is also called the *start node*, a sink is called an *end node*, $\ell(x)$ is called the *logic* of x. If the logic is And, Or or Xor, we call x a *gateway*; if x has no logic and x is no start or end node, we call x a *task*. We use BPMN to depict workflow graphs, i.e., gateways are drawn as diamonds, where the symbol "+" inside stands for And, a circle stands for Or, whereas no decoration stands for Xor. Tasks are drawn as rectangles, start and end nodes as circles. A gateway that has more than one incoming edge and only one outgoing edge is also called a *join*, a gateway with more than one outgoing but only one incoming edge is also called a *split*. We may assume for simplicity of the presentation that every gateway is either a split or a join. We say that an edge e is *incident* to a node n if e is incoming to n or outgoing from n. Let x, y be two graph elements, i.e., nodes or edges. If there is a path from x to y, we also say sometimes that x is *upstream* of y and y is *downstream* of x. If the graph is acyclic, then 'upstream' is a partial order and we write $x \leq y$.

The semantics of a workflow graph is, similarly to Petri nets, defined as a token game. A *state* of a workflow graph is represented by tokens on the edges of the graph. Let $G = (V, E, \ell)$ be a workflow graph. A *state* of G is a mapping $s : E \to \mathbb{N}$, which assigns a natural number to each edge. When $s(e) = k$, we say that edge e carries k *tokens* in state s. The semantics of the various nodes is defined as usual. An And-gateway removes one token from each of its ingoing edges and adds one token to each of its outgoing edges. An Xor-gateway nondeterministically chooses one of its incoming edges on which there is at least one token, removes one token from that edge, then nondeterministically chooses one of its outgoing edges, and adds one token to that outgoing edge. As usual, we abstract from the data that controls the flow in Xor-gateways, hence the nondeterministic choice. Enabledness of an Or-gateway will be defined later in this paper. When an Or-gateway executes, it consumes a token from each incoming edge that carries a token and produces a token for each edge of a nonempty subset of its outgoing edges. That subset is chosen nondeterministically, again abstracting from data-based decisions.

[1] We show some example workflow graphs that have multiple edges from one node to another. The reader may imagine a task on such edges to comply with this formalization.

To be more precise, let s and s' be two states and n a node that is neither a start nor an end node. At this point, we assume that a definition of when an Or-join is *enabled* in a state s of the workflow graph is given, i.e., the following is parametrized by such a definition. Because we will introduce several such definitions later in this paper, we say also an Or-join is X-*enabled* in a state s to make the parameter more explicit. The parameter X will occur in the notions that depend on X-enabledness of Or-joins.

We write $s \xrightarrow{n} s'$ when s changes to s' by executing node n. We have $s \xrightarrow{n} s'$ if

1. $\ell(n) =$ And or the logic of n is undefined, and
$$s'(e) = \begin{cases} s(e) - 1 & e \text{ is an incoming edge of } n, \\ s(e) + 1 & e \text{ is an outgoing edge of } n, \\ s(e) & \text{otherwise.} \end{cases}$$
Note that our assumption that every gateway is either a join or a split implies that an edge cannot be both, incoming and outgoing for the same node.
2. $\ell(n) =$ Xor and there exists an incoming edge e' and an outgoing edge e'' of n such that
$$s'(e) = \begin{cases} s(e) - 1 & e = e', \\ s(e) + 1 & e = e'', \\ s(e) & \text{otherwise.} \end{cases}$$
3. $\ell(n) =$ Or, x is X-enabled in s, and there exists a nonempty set F of outgoing edges of n such that
$$s'(e) = \begin{cases} s(e) - 1 & e \text{ is an incoming edge of } n \text{ such that } s(e) \geq 1, \\ s(e) + 1 & e \in F, \\ s(e) & \text{otherwise.} \end{cases}$$

The *initial state* of G is the state where there is exactly one token on the unique outgoing edge of the start node and no token anywhere else. A node n that has Xor- or And-logic or is an Or-split is *enabled* in a state s if there exists a state s' such that $s \xrightarrow{n} s'$. We also say X-*enabled* in a context where n can be an Or-join or another gateway. A state s' is X-*reachable from* a state s if there exists a finite sequence $s_0 \xrightarrow{n_1} s_1 \ldots s_{k-1} \xrightarrow{n_k} s_k, k \geq 0$ such that $s_0 = s$ and $s_k = s'$. Such a sequence is called an X-*execution*. A state is a X-*reachable* state of G if it is reachable from the initial state of G.

An X-reachable state s is a (*local*) X-*deadlock* if there exists a token on an incoming edge of a gateway such that every state that is X-reachable from s also contains a token on that edge. A state is *unsafe* or has a *lack of synchronization* if there is an edge which carries more than one token in s, otherwise it is *safe*. A workflow graph G is X-*live* if it has no X-deadlock and X-*safe* if no X-reachable state has a lack of synchronization. G is X-*sound* if it is X-live and X-safe.

Note that all notions are independent of the parameter X in case G does not contain an Or-join.

3 Semantics for Acyclic Workflow Graphs

To clarify the Or-join semantics for acyclic graphs, we define a state-space based and a graph-based semantics of the Or-join in this section. We show that both coincide on sound workflow graphs. Furthermore, we show that the graph-based semantics is the same that is induced by the execution semantics for BPEL flows, i.e., *dead path elimination*.

It is agreed that at least one incoming edge of the Or-join needs to have a token for its enabledness. It is also usually assumed that, if there is a token on each incoming edge, the Or-join is enabled. We are now going to interpret what it means that the Or-join has to wait for any tokens that 'may still arrive' on the empty incoming edges.

3.1 The state-space view

The problem of formalizing the state-space based interpretation of 'a token may arrive' arises already for acyclic workflow graphs. Given a state s of the graph, a token *may arrive* on an edge e if s can evolve into a state s' such that e has a token in s'. However, before it can be defined whether a state can evolve into another state, we need to define whether an Or-join is enabled. Conversely, before we can define whether an Or-join is enabled, we need to define when a state can evolve into another state. This cyclic dependency prevents a straight-forward formalization. To avoid fixed point theory as a resolution here, we can exploit the fact that, in an acyclic workflow graph, the dependencies between multiple Or-joins are given by the partial order that is induced by the edges of the graph. In particular, minimal Or-joins with respect to this order do not depend on any other Or-join and their semantics can be defined without referring to the Or-join semantics:

Definition 1. *Let G be an acyclic workflow graph.*

1. *Let j be an Or-join of G. The* depth *of j is the largest number of Or-joins that are contained on any path from the start node to j (not counting j itself).*
2. *We say that a state s' is* k-S-reachable *('S' stands for 'state-based') from a state s for $k \geq 0$ if there is an S-execution that starts in s, ends in s', and that does not contain any Or-join of depth $\geq k$.*
3. *An Or-join of depth k is* S-enabled *in a state s for $k \geq 0$ if $s(e) \geq 1$ for some incoming edge e of x and if for each incoming edge e' of x with $s(e') = 0$, there is no state s' such that s' is k-S-reachable from s and $s'(e') \geq 1$.*

To evaluate S-enabledness of an Or-join according to this definition, one can first evaluate the S-enabledness of Or-joins of depth 0, then depth 1 and so forth. To see whether an Or-join of depth 0 is S-enabled, one has to check whether a token can be produced on an empty incoming edge of the Or-join by executing only Or-splits, And- and Xor-gateways but not Or-joins. To see whether an Or-join of depth 1 is S-enabled, one has to check whether a token can be produced on an empty incoming edge of the Or-join by executing only Or-splits, And- and Xor-gateways and Or-joins of depth 0. Consider for example Fig. 3. In part (a), $j1$ has depth 0 and is S-enabled, whereas $j2$ has depth 1 and is not S-enabled. In part (b), both $j1$ (depth 0) and $j2$ (depth 1) are S-enabled.

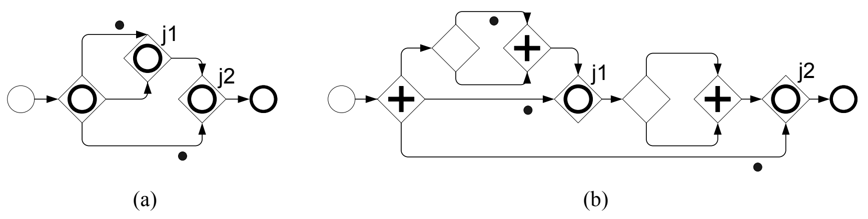


Fig. 3. Examples for the state-space and graph-based interpretation

3.2 The graph-based view

An alternative interpretation of 'may arrive' is graph-based and was indicated in the BPMN 1.0 specification [5][2]: A token may arrive on an edge e if there is an edge e' that has a token in s and there is a directed path from e' to e in the workflow graph. This interpretation has the advantage of a straight-forward formalization and simple and efficient enactment:

Definition 2. *Let G be an acyclic workflow graph. An Or-join j is* P-enabled *('P' stands for 'path') in a state s if* $s(e) \geq 1$ *for some incoming edge e of j and if for each incoming edge* e' *of j with* $s(e') = 0$ *and for each edge* e'' *of the graph with* $s(e'') \geq 1$*, there is no directed path from* e'' *to* e'*.*

In Fig. 3(a), $j1$ is P-enabled, whereas $j2$ is not. In Fig. 3(b), neither $j1$ nor $j2$ is P-enabled. Hence, S-enabledness and P-enabledness coincide in Fig. 3(a) but they differ in Fig. 3(b). The reason is that the graph in Fig. 3(b) contains deadlocks. The deadlocks prevent tokens to move to the empty incoming edges of the Or-joins. While the state-space interpretation is aware of the deadlocks, the graph-based interpretation is not. Since such deadlocks are usually considered modeling errors, the difference of the two semantics on such models is not substantial. In fact, we can show that the state-space and graph-based interpretation coincide on sound acyclic graphs:

Definition 3. *Given a workflow graph G, we say that two semantics X and Y* coincide under soundness *if the following statements hold:*

1. *G is X-sound if and only if it is Y-sound.*
2. *If G is X-sound (or equivalently Y-sound), then a state is X-reachable whenever it is Y-reachable.*
3. *If G is X-sound (or equivalently Y-sound), then an Or-join is X-enabled in an X-reachable (equivalently: Y-reachable) state whenever it is Y-enabled.*

Theorem 1. *P-semantics and S-semantics coincide under soundness on acyclic workflow graphs.*

[2] Dumas *et al.*[2] have proposed a different graph-based interpretation.

Proof. We sketch here the main idea. A more detailed proof is included in the appendix: If an Or-join j is not P-enabled because there is a token upstream of an empty incoming edge of j, then it can be shown that that token can be moved to that empty incoming edge of j, provided the graph is deadlock-free. Then, j is also not S-enabled.

Conversely, if j is not S-enabled because a token can be brought to an empty incoming edge of j, then that token must be located upstream of the empty incoming edge of j and hence j is not P-enabled.

Note that P-semantics can be enacted in linear time in the number of edges of the graph, i.e., it can be determined in linear time whether an Or-join is enabled in a given state.

3.3 Dead path elimination

In this section, we show that P-semantics essentially coincides with semantics that is implied by *Dead Path Elimination* (DPE) for BPEL flows. Dead path elimination determines the enabledness of an Or-join in an acyclic graph by inspection of its incoming edges only. This works as follows. Let G be an acyclic workflow graph. A *DPE-state* is again a distribution of tokens over the edges of G, however each token has now a value, which is either *true* or *false*. Initially there is a *true* token on the initial edge and no token elsewhere. Each gateway waits for a token, *true* or *false*, on each incoming edge. As soon as all incoming edges have a token, these tokens are consumed and a token is produced on each outgoing edge. If all inputs are false, all outputs are false as well. Upon receipt of a true token, an Or-split produces *true* tokens on a nonempty subset of outgoing edges and *false* tokens on all other edges. The Xor-split and the And-split can be thought of as special cases of the Or-split: the former always produces exactly one *true* token, whereas the latter always produces only *true* tokens. The value of the token that is produced by a join is determined by applying the logical function of the join to the inputs, i.e., an Or-join applies the Or-function, the And-join the And-function and the Xor-join the Xor-function.

Figure 4 shows an example of dead path elimination. The labeling represents an entire DPE execution. An intermediate reachable DPE state is represented by the tokens shown in italics. The following lemma characterizes reachable DPE states. It can be easily shown by induction on the reachable DPE states.

Lemma 1. *Let G be acyclic. Then for each reachable DPE-state x and each edge e of G, exactly one of the following three statements is true: (1) e has a token in x, (2) some edge $e' < e$ has a token or (3) some edge $e' > e$ has a token.*

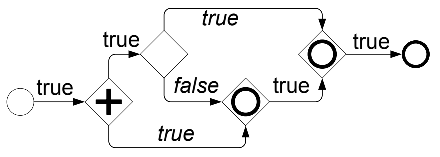


Fig. 4. An example for dead path elimination

We say that a DPE step is a *true* step, denoted $x \xrightarrow{n} x'$, if at least one true token is consumed, otherwise we say it is an *elimination* step, denoted $x \overset{n}{\dashrightarrow} x'$. If x' can be reached from x through elimination steps only (zero or more), we write $x \overset{*}{\dashrightarrow} x'$. Furthermore, we write $x \overset{\max}{\dashrightarrow} x^*$ if $x \overset{*}{\dashrightarrow} x^*$ and x^* does not enable any further elimination step. It is not difficult to see that x^* always exists and is unique. We say that a step $x \overset{n}{\dashrightarrow} x'$ is *unsound* if either n is an And-join that consumes at least one false token in this step or n is an Xor-join that consumes more than one true token in this step. We say that the graph G is *DPE-sound* if no reachable DPE-state enables an unsound step. Note that if the graph is DPE-sound, all Xor- and And-joins can be replaced by Or-joins without changing the behavior.

We show now that for sound acyclic graphs, P-execution and DPE execution mutually simulate each other in a strong sense. To this end, we say that a safe P-reachable state s and a reachable DPE-state x *correspond*, denoted $s \sim x$, if for all edges e, s has a token on e if and only if x has a true token on e. Thus a state may correspond to more than one DPE-state. DPE semantics and P-semantics coincide if elimination steps are hidden. This can be formalized as follows:

Theorem 2. *Let G be an acyclic workflow graph, s be P-reachable state, x a reachable DPE-state and n a node of G.*

1. *The initial state and the initial DPE-state of G correspond.*
2. *Let $x \xrightarrow{n} x'$ be a sound DPE step and $s \sim x$. Then, there exists a state s' such that $s \xrightarrow{n} s'$ and $s' \sim x'$.*
3. *$s \sim x$ and $x \overset{*}{\dashrightarrow} x'$ implies $s \sim x'$,*
4. *Let G be P-safe. Then, $s \sim x$, $s \xrightarrow{n} s'$ and $x \overset{max}{\dashrightarrow} x^*$ implies that there exists a DPE state x' such that $x^* \xrightarrow{n} x'$ and $s' \sim x'$.*
5. *G is P-sound if and only if G is DPE-sound.*

Proof. Again, we sketch here the main ideas. A more detailed proof is included in the appendix: For part 2, where $s \sim x$, one can show by help of Lemma 1 that a false token on an edge e in x implies that there is not only no token on e in s but also no token in s on any edge upstream from e. It immediately follows that an OR-join that is enabled in x is also enabled in s.

For part 4, consider an Or-join n that is P-enabled in s. If n has an empty incoming edge e in s, it can be shown, again by help of Lemma 1, that there must be a false token upstream of e in x. Because false tokens move as far as possible when going from x to x^*, it can be shown that the Or-join is then enabled in x^*.

4 Semantics for Cyclic Workflow Graphs

In this section, we extend the graph-based semantics to deal with cyclic workflow graphs. While the graph-based semantics addresses the first fundamental problem mentioned in Sect. 1, we encounter the second problem mentioned in Sect. 1 when we want to extend the semantics to cyclic graphs: Figure 5 shows an example of a graph with two mutually dependent Or-joins, which is known as a ‘vicious circle’. Applying a

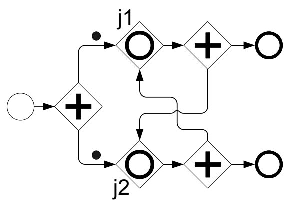


Fig. 5. A symmetric vicious circle

state-space interpretation, $j1$ is enabled if and only if $j2$ is not enabled and $j2$ is enabled if and only if $j1$ is not enabled. If we apply the graph-based interpretation as defined in Def. 2, then the state in Fig. 5 is a deadlock. This may be satisfactory because it is not clear what the intended behavior of the graph in Fig. 5 is and hence what semantics it should have. As it defines a deadlock, it can be sorted out by static analysis as a modeling error. However, we show in the next subsection that this argument does not apply to all vicious circles.

4.1 Block-based semantics for separable graphs

Although, it seems that the example in Fig. 5 does not present a serious problem, there are more natural workflow graphs that exhibit the same problem. Figure 6 shows a *well-structured* workflow graph, i.e., it is composed of matching pairs of splits and joins. In contrast to the previous example in Fig. 5, its not difficult to imagine a real business process behind it—provided that suitable tasks are inserted at the edges of the graph. However, the informal semantics produces the same problems as for the example in Fig. 5: The join $j2$ should wait for the token at $e1$ to arrive at $e4$ and the join $j1$ should wait for the token at $e3$ to arrive at $e2$, which can go there via $s3$ and $j3$. This is also reflected by the graph-based semantics as in Def. 2, now applied to a cyclic graph, which interprets the state shown as a deadlock. Previous semantic proposals [8, 3, 2] either define this as a deadlock (neither $j1$ nor $j2$ is enabled) or a nondeterministic choice between the two joins (both, $j1$ and $j2$ are enabled but execution of one disables the other). Moreover, note that a execution of $j2$ implies that the graph is then unsound.

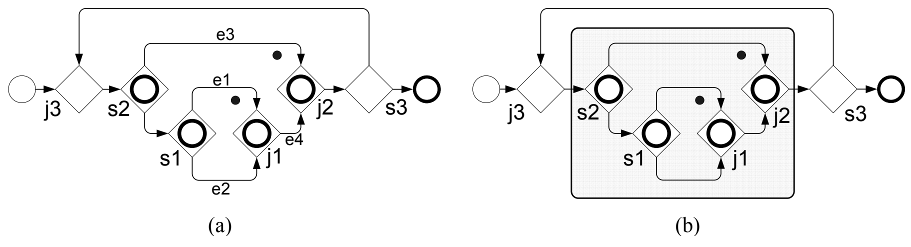


Fig. 6. A vicious circle in a well-structured graph

We believe that neither of these two options is reasonable for this graph. It is a well-structured graph, in fact it can be thought of a BPMN representation of a BPEL process where a *flow*, i.e., a sound acyclic subgraph, is nested in a repeat-until-loop, cf. Fig. 6(b). BPEL semantics implies a block-based execution: Before a new iteration of the loop is started, the flow has to complete first. This means that $j2$ waits for $j1$ but not the other way around, i.e., the structure of the process implies an asymmetric dependency between $j1$ and $j2$. We believe that a block-based execution as in BPEL provides the natural semantics for such well-structured graphs. This was also advocated by Mendling and van der Aalst [4].

To capture this semantics, we define a class of workflow graphs, called *separable graphs*, which include well-structured graphs. A *separable graph* is a workflow graph that can be composed from sound acyclic and from sequential *blocks*:

Definition 4. *Let G be a workflow graph with nodes N and edges E such that G has a unique end node n_ω and let n_α be its unique start node.*

1. *Let $B \subseteq N$ be a nonempty set of nodes and $E_B = E \cap (B \times B)$ the induced set of edges. B is a* block *if the induced subgraph (B, E_B) has a single entry edge and a single exit edge, i.e., there exist edges $e, e' \in E$ with $E \cap ((N \setminus B) \times B) = \{e\}$ and $E \cap (B \times (N \setminus B)) = \{e'\}$; e and e' are called the* entry *and the* exit *edge of B, respectively.*
2. *Let $B_0 = N \setminus \{n_\alpha, n_\omega\}$. B_0 is a block, called the* root block. *A* decomposition *of G is a set $\mathscr{D}$ of blocks of G such that $B_0 \in \mathscr{D}$ and for each pair of blocks $B, B' \in \mathscr{D}$, we have $B \subseteq B'$, $B' \subseteq B$, or $B \cap B' = \emptyset$. If $B' \subseteq B$, we say B' is a* subblock *of B. B' is an* immediate subblock *of B if there is no block $B'' \in \mathscr{D}$ such that $B' \subset B'' \subset B$.*
3. *Let $B \in \mathscr{D}$ be a block. The* abstraction *of B with respect to $\mathscr{D}$ is the workflow graph that results from taking the graph (B, E_B), replacing each immediate subblock $B' \in \mathscr{D}$ of B by a fresh task and adding a fresh start node and a fresh end node that are connected to the entry and the exit of the block respectively. B is* sequential *if each gateway in its abstraction has Xor logic; B is a* flow *if its abstraction is acyclic and sound. G is* separable *if there is a decomposition $\mathscr{D}$ such that each block in $\mathscr{D}$ is either a flow or sequential; $\mathscr{D}$ is then called a* separating *decomposition of G.*

Definition 5. *Let G be a separable workflow graph and $\mathscr{D}$ a separating decomposition of G. An Or-join j is $\mathscr{D}$-enabled in a state s if it is P-enabled in s with respect to the smallest block in $\mathscr{D}$ that contains j.*

Note that in a separable workflow graph, the smallest block that contains an Or-join is not sequential and hence must be acyclic.

Lemma 2. *Let G be a separable workflow graph and $\mathscr{D}$ a separating decomposition. Then G is $\mathscr{D}$-sound.*

Proof. We say that a block is *active* in a state s of G if some edge $e \in E_B$ of the block has a token in s. It can be shown by structural induction on $\mathscr{D}$ that a block has no deadlock and lack of synchronization provided that its subblocks are sound and provided that its entry edge never gets a token while the block is active. Since the top-level block, i.e., the workflow graph G itself, never gets more than one token, it follows that G is $\mathscr{D}$-sound.

$\mathscr{D}$-semantics does not seem satisfactory since a decomposition is used to define it. However, the semantics does not depend on the particular decomposition. For example, the separating decomposition $\mathscr{D}_1 = \{B_0, B_1\}$ shown in Fig. 6(b), where $B_0 = \{s1, j1, s2, j2, s3, j3\}$ and $B_1 = \{s1, j1, s2, j2\}$ induces the same semantics as $\mathscr{D}_2 = \{B_0, B_1, B_2\}$ where $B_2 = \{s1, j1\}$.

Proposition 1. *Let $\mathscr{D}$ and $\mathscr{D}'$ be two separating decompositions of G. Then, $\mathscr{D}$-semantics and $\mathscr{D}'$-semantics coincide on G.*

Proof. The main idea is the following. (A more detailed proof is included in the appendix.) The essential proof obligation is to show that an Or-join j is enabled with respect to a containing block X whenever it is enabled with respect to another containing block Y. It can be shown that one can restrict to the case $Y \subseteq X$. One direction is trivial. For the other direction, suppose that j is enabled in Y but not in X. Then, there is a token upstream of j in $X \setminus Y$. Because there is no deadlock (Lemma 2), it can be shown that we can then move the token to the entry of Y and then further to the non-empty incoming edge of j, which would cause a lack of synchronization, which contradicts Lemma 2.

Although this block-based semantics gives a reasonable meaning to separable graphs that is aligned with block-based execution as in BPEL, its description is not yet fully satisfactory because one needs to find a decomposition in order to figure out the enabledness of an Or-join. We will see later that it is not necessary to be aware of the blocks. The same semantics can be defined in a different way that does not refer to blocks.

4.2 A new semantics

We consider again the well-structured graph in Fig. 6. The block-based semantics suggests that the dependency between the joins $j1$ and $j2$ is not as symmetric as in Fig. 5. The structure suggests that $j2$ should wait for $j1$ but not vice versa. How can we characterize this asymmetry? Let us compare the path from $e1$ to $e4$ with the path from $e3$ to $e2$. The path from $e1$ to $e4$ shows how the token on $e1$ catches up to its 'sibling token' on $e3$. However, if we move the token from $e3$ to $e2$, then this starts a new iteration of the loop, causing the synchronization of two tokens from different rounds. Starting a new round is witnessed for example by the fact that the path from $e3$ to $e2$ visits the gateway $s1$ because $s1$ has also a path to $e1$ where there is already a token waiting. Executing $s1$ therefore could cause a second token on $e1$ which would manifest the synchronization error. Clearly, such a new iteration would also be started if the path visits the waiting join itself. For example, the join $j1$ should not wait for the token on $e1$ to arrive at $e2$. Any corresponding path from $e1$ to $e2$ goes through $j1$ and should therefore not be considered. We therefore define the semantics as follows:

Definition 6. *An Or-join j is* Q-enabled *in state s if*

1. *there is an incoming edge e of j such that $s(e) \geq 1$ and*

2. *for each edge e' of the graph with $s(e') \geq 1$, we have: If there is a path from e' to some incoming edge e of j with $s(e) = 0$ that does not visit j, then there is a path from e' to some incoming edge e of j with $s(e) > 0$ that does not visit j.*

This definition implicitly defines *inhibiting paths* to j, i.e., a path from a token an empty incoming edge of j such that the path does not visit j. Furthermore, an *anti-inhibiting path* to j is a path from a token to a non-empty incoming edge of j such that the path does not visit j. An Or-join j has to wait only for those token that have an inhibiting path but no anti-inhibiting path to j. Note that there is no anti-inhibiting path in Fig. 6(a) to $j2$ from the token on $e1$ because the path from $e1$ to $e3$ visits $j2$.

We show now that Q-semantics and P-semantics coincide on sound acyclic graphs.

Proposition 2. *P-semantics and Q-semantics coincide on P-sound acyclic workflow graphs.*

Proof. See Appendix.

Theorem 3. *Let G be a separable workflow graph and $\mathscr{D}$ a separating decomposition. Then $\mathscr{D}$-semantics and Q-semantics coincide on G.*

Proof. See Appendix.

Note that the compliance with block-based semantics is related to the fact that the Q-enabledness of an Or-join j does not depend on the tokens outside the smallest block that contains j. In other words, the Or-join behavior is non-local only within that block. This remains true for non-separable graphs under a mild syntactic restriction[3]. This locality of Q-semantics with respect to such a block makes the behavior of a larger graph easier to understand provided it contains smaller blocks. Moreover, this means that such blocks can be understood, executed and analyzed in isolation, i.e., they constitute logically atomic parts of the workflow graph in Q-semantics. Furthermore, refining a task with such a block or replacing a block with another block does not change the behavior of the surrounding graph. This eases constructing, changing and refactoring a workflow graph. In particular, the behavior of a business process does not change when a process model is changed by encapsulating such a block into a subprocess.

4.3 Non-separable graphs

We have argued in the previous section that Q-semantics suits separable graphs well. In this section, we study examples of how non-separable graphs behave under Q-semantics.

We first look at some examples, for which we think that the intended meaning is clear and also achieved by Q-semantics. Figure 7 shows two cyclic graphs where an Or-region, i.e., a pair of an Or-split and an Or-join, has an additional entry on one of the paths between the split and join. Figure 8(a) can be seen as an Or-region with two parallel additional entries. Figure 7(a) and (b) can be seen as special cases of Fig. 8(a) where one additional entry was omitted. The correspondence between Fig. 7(a) and

[3] One has to require that for every incoming edge e of an Or-join j contained in the block, there is a path from the start node to e that does not visit j.

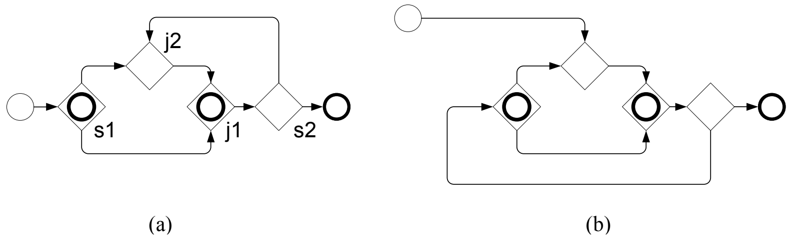


Fig. 7. Two non-separable cyclic graphs

Fig. 8(a) is given by the labeling of the gateways. Figure 8(b) shows how an Or-join can be used to glue a flow with a well-structured loop. All these examples are Q-sound. The reader may verify that Q-semantics indeed produces the desired behavior for these examples.

The symmetric vicious circle in Fig. 5 still creates a deadlock in Q-semantics. As we argued before, we do not consider this as a problem. The intended behavior is not clear and therefore, the graph can be rejected by static analysis.

Figure 9(a) shows a graph where the situation seems to be different at first sight. The dependency between the two Or-joins *j*1 and *j*2 does not seem to be symmetric. In fact, the graph is similar to the graph in Fig. 7(a), they only differ in the logic of *j*2. However, Q-semantics defines the state shown as a deadlock. This may contrast our intuition because one might think that this graph should have the same behavior as the graph in Fig. 7(a), so that *j*1 waits for *j*2 but not the other way around. However, this intuition may be caused by the layout of the graph: Fig. 9(b) shows how the same graph can be re-drawn. Now, the layout suggests that *j*2 should wait for *j*1 and not the other way around. Thus, the dependency is more symmetric than these two layouts suggest and in fact, the graph can also be re-drawn as shown in Fig. 9(c) to resemble more the graph in Fig. 5. Again, we think that the intended behavior is not clear and that the deadlock can be used to reject the graph in static analysis.

How to treat such 'vicious circles' in practice can be seen as a meta-issue, which is fairly independent from the discussion in this paper. The deadlocks produced by these examples can be detected at analysis time or at runtime (cf. [2]) and they can be

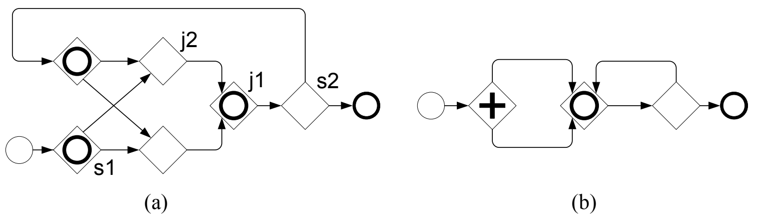
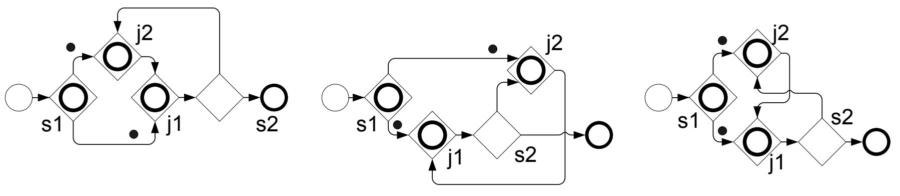


Fig. 8. Two more sound cyclic graphs



(a) (b) (c)

Fig. 9. Three different layouts of the same cyclic graph

treated in a different way by an engine or tool as defined through a 'meta-semantics'. In particular, an engine could apply a more *optimistic approach* [3, 2] and resolve the deadlocks through a non-deterministic choice between the Or-joins if there are reasons to do so. This approach clearly remains possible on top of Q-semantics.

4.4 Enactment

In this section, we show that Q-semantics of an Or-join can be enacted in linear time.

Procedure 1 Returns *true* iff Or-join j is Q-enabled in state s.

```
IsEnabled(Workflow graph G, State s, Or-join j)
  Red := $\{e \mid e$ is an incoming edge of $j$ such that $s(e) > 0\}$
  while there exist an edge $e = (n_1, n_2) \in$ Red and $e' = (n_3, n_1) \notin$ Red such that $n_1 \neq j$ do
    Red := Red $\cup \{e'\}$
  Green := $\{e \mid e$ is an incoming edge of $j$ such that $s(e) = 0\}$
  while there exist an edge $e = (n_1, n_2) \in$ Green and $e' = (n_3, n_1) \notin$ (Green $\cup$ Red) such that $n_1 \neq j$ do
    Green := Green $\cup \{e'\}$
  return (Green $\cap \{e \mid s(e) > 0\} = \emptyset$).
```

If an Or-join j has at least one token on an incoming edge, we have to determine whether there are any inhibiting and anti-inhibiting paths to j. The algorithm consists of two parts. First we mark all edges of the graph that contribute to anti-inhibiting paths. Those can be determined by backward reachability search starting from the non-empty incoming edges of j. We mark all those edges in red. We stop exploration when we reach j itself in order not to mark the empty incoming edges of j. The second part of the algorithm marks all edges in green that are not red already and that are backward-reachable from any empty incoming edge of j. Again, we stop exploration when we reach j itself in order not to mark any non-empty incoming edge of j. This part computes the inhibiting paths. The Or-join j is then enabled if and only if there is no token on some green edge. The Pseudo-code in Procedure 1 makes this algorithm more precise. It is not difficult to see that this algorithm can be implemented to run in linear time in the size of the workflow graph. We conclude:

Theorem 4. *Let G be a workflow graph. It can be computed in linear time (in the size of G) whether an Or-join is Q-enabled in a given state of G.*

5 Conclusion

We have presented a new semantics for the Or-join in workflow graphs. We have argued that a graph-based semantics gives rise to a straight-forward formalization as well as an efficient enactment. Furthermore, we have pointed out that reasonable process graphs exist where 'vicious circles' arise and argued that those should neither result in a deadlock nor be resolved by non-deterministic choice. We have shown that a natural semantics can be defined for those cases by extending the graph-based semantics for acyclic graphs to general workflow graphs. Our semantics is aligned with block-based execution of well-structured graphs. We have shown various examples of cyclic graphs that are not well-structured but still receive an intuitive semantics in our proposal. Nevertheless, there are cases where mutually dependent Or-joins create a deadlock in our semantics. We have argued that in those cases, the intended meaning is not clear and hence they should be sorted out by static analysis.

As usual, unsoundness is considered as a modeling error in this paper. In fact, lack of synchronization can cause undesired race conditions for Or-join enabledness in Q-semantics. Extending our semantics to models which treat lack of synchronization as a feature rather than an error is beyond the scope of this paper.

Our semantics is included in the draft for the BPMN 2.0 standard [6].

Acknowledgement. We would like to thank Alistair Barros and the other members of the BPMN 2.0 RFP taskforce, in particular Matthias Kloppmann and Stephen A. White, moreover we thank Ekkart Kindler, Jussi Vanhatalo and Thomas Hettel for helpful discussions. We also would like to thank the anonymous reviewers for their constructive comments to improve the presentation.

References

1. Egon Börger, Ole Sörensen, and Bernhard Thalheim. On defining the behavior of OR-joins in business process models. *J. UCS*, 15(1):3–32, 2009.
2. Marlon Dumas, Alexander Großkopf, Thomas Hettel, and Moe Thandar Wynn. Semantics of standard process models with OR-joins. In *OTM Conferences (1)*, *LNCS* 4803, pages 41–58. Springer, 2007.
3. Ekkart Kindler. On the semantics of EPCs: Resolving the vicious circle. *Data Knowl. Eng.*, 56(1):23–40, 2006.
4. Jan Mendling and Wil M. P. van der Aalst. Formalization and verification of EPCs with OR-joins based on state and context. In *CAiSE*, *LNCS* 4495, pages 439–453. Springer, 2007.
5. OMG. Business process modeling notation (BPMN) version 1.0, OMG document number dtc/06-02-01. Technical report, 2006.
6. OMG. Business process model and notation (BPMN) version 2.0, OMG document number dtc/2010-05-03. Technical report, 2010.
7. Wil M. P. van der Aalst, Jörg Desel, and Ekkart Kindler. On the semantics of EPCs: A vicious circle. In *EPK*, pages 71–79. GI-Arbeitskreis Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, 2002.

8. Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.
9. Moe Thandar Wynn, David Edmond, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede. Achieving a general, formal and decidable approach to the OR-join in workflow using reset nets. In *ICATPN*, *LNCS* 3536, pages 423–443. Springer, 2005.

Appendix: Proofs

This appendix provides outlines of some proofs that were omitted in the main text of this paper.

5.1 Proof of Thm. 1

Theorem 1. *P-semantics and S-semantics coincide under soundness on acyclic workflow graphs.*

We need the following Lemma:

Lemma 3. *Let $X \in \{S,P\}$. Let G be an acyclic, X-live workflow graph, e, e' edges and s an X-reachable state of G such that $s(e) \geq 1$ and $e \leq e'$. Then there exists a state s' that is X-reachable from s such that $s'(e') \geq 1$.*

Proof. Let the *depth* of an edge e be the length of the longest path leading to e. The proof is a straight-forward induction on the depth of e'.

Proof. (of Theorem 1) Let G be an acyclic workflow graph. We break the claims down into the following steps:

1. Let s be a state and j an Or-join of G. If j is P-enabled in s then it is also S-enabled. *Proof:* Suppose not. Then there is a state s' that is k-S-reachable from s, where k is the depth of j, such that there is a token on an incoming edge of j in s' that is empty in s. Such a token can be traced back to a token in s that is located upstream of the incoming edge of j that is empty in s, which contradicts j being P-enabled in s.
2. Let G be S-live and s an S-reachable state of G. If j is S-enabled in s then it is also P-enabled in s. *Proof:* Suppose not. Then there is a token upstream of an empty incoming edge of j. Since G is S-live, there is an S-execution that brings that token to the empty incoming edge of j (by Lemma 3), which contradicts j being S-enabled in s.
3. If a state s is P-reachable then it is S-reachable. *Proof:* By induction over P-executions exploiting claim 1 of this proof.
4. Let G be S-live. If a state s is S-reachable, then it is P-reachable. *Proof:* By induction over S-executions exploiting claim 2.
5. If G is S-live, then it is also P-live. *Proof:* Suppose not. Then G has a P-deadlock. Consider a minimal prefix G' of G exhibiting a P-deadlock, and let s be a maximal (with respect to P-reachability) deadlock of G'. Say a token is blocked at a join j. It follows that there is no token upstream of an empty incoming edge of j. Hence j cannot be an Or-join and must be an And-join. Because s is S-reachable according to claim 3, s is also an S-deadlock, which contradicts our assumption that G is S-live.
6. If G is P-live, then it is also S-live. *Proof:* Similar to the previous claim.
7. G is S-sound if and only if G is P-sound. *Proof:* If G is S-live then it is also P-live, moreover reachability coincides and therefore unsafeness coincides as well. If G is P-live then it is S-live, hence reachability and unsafeness must again coincide.

All claims of the theorem are now established.

5.2 Proof of Thm. 2

Theorem 2. *Let G be an acyclic workflow graph, s be P-reachable state, x a reachable DPE-state and n a node of G.*

1. *The initial state and the initial DPE-state of G correspond.*
2. *Let $x \xrightarrow{n} x'$ be a sound DPE step and $s \sim x$. Then, there exists a state s' such that $s \xrightarrow{n} s'$ and $s' \sim x'$.*
3. *$s \sim x$ and $x \overset{*}{\dashrightarrow} x'$ implies $s \sim x'$,*
4. *Let G be P-safe. Then, $s \sim x$, $s \xrightarrow{n} s'$ and $x \overset{max}{\dashrightarrow} x^*$ implies that there exists a DPE state x' such that $x^* \xrightarrow{n} x'$ and $s' \sim x'$.*
5. *G is P-sound if and only if G is DPE-sound.*

Proof. Claims 1 and 3 are trivial. For claim 2, the only interesting case is that n is an Or-join. Because a true step is enabled with respect to n, there is at least one true token on an incoming edge of n and all other incoming edges have tokens of either value. Lemma 1 implies that a token on an edge e in a reachable DPE state implies that all edges that are upstream of e are empty. Since s corresponds to x, n has at least one incoming token and all edges that are upstream of empty incoming edges of n are empty as well. Hence n is enabled in s.

We now show claim 4. It is easy to check that $s \sim x$, $s \xrightarrow{n} s'$ and $x \xrightarrow{n} x'$ implies $s' \sim x'$. It remains to be shown that n is enabled in x^*. It is clear that this is true if n is an And-join. Suppose n is an Xor-join or an Or-join that is not enabled in x^*. Then some incoming edge e of n does not have a token in x^*. It follows from Lemma 1 that there is no token downstream of e because some incoming edge of n has a token. Hence there is a token upstream of e. That token cannot be a true token. (If n is an Or-join, this would contradict n being enabled in s, if n is an Xor-join, we could use claims 1 and 2 to produce a P-reachable lack of synchronization) Therefore the token must be a false token. Then there is a minimal node that has one false token as input. It follows that all inputs of that node carry false tokens (again due to Lemma 1). Then an elimination step is enabled in x^*, which contradicts our assumption.

To prove claim 5, suppose that G is P-sound but not DPE-sound. Consider a DPE execution that ends in a DPE-state x that enables an unsound step $x \xrightarrow{n} x'$. We use claims 1 and 2 to simulate this DPE-execution by P-execution and obtain a state s that corrsponds to x. In case n is an And-join, it follows that s is a P-deadlock because there is no token in s upstream the incoming edge of n that has a false token in x. If n is an Xor-join, we have a token in more than one incoming edge to n in s and can produce a lack of synchronization. Both cases contradict P-soundness. The converse direction is similar.

5.3 Proof of Prop. 1

Proposition 1. *Let $\mathscr{D}$ and $\mathscr{D}'$ be two separating decompositions of G. Then, $\mathscr{D}$-semantics and $\mathscr{D}'$-semantics coincide on G.*

Proof. We break the proof down into the following claims:

1. An Or-join j is $\mathscr{D}$-enabled in a state s if and only if j is $\mathscr{D}'$-enabled in s. *Proof:* Let X be the smallest block in $\mathscr{D}$ that contains j and X' be the smallest block in $\mathscr{D}'$ that contains j. Let $Y = X \cap X'$. It can be shown that Y is a block that is contained in X as well as X' and we have $j \in Y$. We prove that j is enabled in Y iff it is enabled in X, and hence by symmetry in X'. Clearly, if j is enabled in X then it must be enabled in Y. Suppose j is enabled in Y but not in X. Then there must be a token in $X \setminus Y$ upstream an empty incoming edge of j. However, since X is P-live, we can move that token to non-empty incoming edge of j, which contradicts P-safeness of X.
2. A state s is $\mathscr{D}$-reachable if and only if s is $\mathscr{D}'$-reachable. *Proof:* follows immediately from claim 1.

5.4 Proofs of Prop. 2 and Thm. 3

Proposition 2. *P-semantics and Q-semantics coincide on P-sound acyclic workflow graphs.*

Proof. Let G be a P-sound acyclic workflow graph, s a state and j an Or-join of G.

1. If j is P-enabled in s, then also Q-enabled. *Proof:* Immediate.
2. If s is P-reachable, then also Q-reachable. *Proof:* Follows directly from claim 1.
3. If s is Q-reachable, then also P-reachable. *Proof:* Suppose not. Consider a Q-step from s to s' such that s is P-reachable but s' is not. This step must be an Or-join step. Because s' is not P-reachable, the Or-join is not P-enabled in s. Because the Or-join is Q-enabled in s, it follows that in s, there is a token upstream of a nonempty incoming edge of the Or-join. However that contradicts s being P-reachable and G being P-sound.
4. If s is Q-reachable and j is Q-enabled in s, then s is also P-enabled. *Proof:* Follows directly from claim 3.
5. G is Q-sound. *Proof:* Follows from P-soundness of G and claims 2 and 3.

Theorem 3. *Let G be separable and $\mathscr{D}$ a separating decomposition. Then $\mathscr{D}$-semantics and Q-semantics coincide on G.*

Proof. 1. G is Q-sound. *Proof:* Similar to the proof of Lemma 2 by induction over $\mathscr{D}$.

2. If j is $\mathscr{D}$-enabled in s then j is Q-enabled in s. *Proof:* Let e be the entry edge of the smallest fragment in $\mathscr{D}$ that contains j. Because j is $\mathscr{D}$-enabled in s, there is no token on any path from e to an empty incoming edge of j. Any other path from a token to an empty incoming edge of j hence visits e and then there exists also a path from that token to the non-empty incoming edge of j. It follows that j is Q-enabled in s.
3. Let s be Q-reachable. If j is Q-enabled in s, then j is also $\mathscr{D}$-enabled in s. *Proof:* Suppose not. Then there is a token and a path from that token to a non-empty incoming edge of j. Applying Lemma 3, which can be easily proven for Q-semantics, we can move the token to the non-empty incoming edge resulting in a lack of synchronization and a contradiction with G being Q-sound.
4. A state s is $\mathscr{D}$-reachable iff s is Q-reachable. *Proof:* Follows from claims 2 and 3.