Towards EPC Semantics based on State and Context

Jan Mendling
WU Vienna
jan.mendling@wu-wien.ac.at

Wil van der Aalst
TU Eindhoven
w.m.p.v.d.aalst@tue.nl

Abstract: The semantics of the OR-join have been discussed for some time, in the context of EPCs, but also in the context of other business process modeling languages like YAWL. In this paper, we show that the existing solutions are not satisfactory from the intuition of the modeler. Furthermore, we present a novel approach towards the definition of EPC semantics based on state and context. The approach uses two types of annotations for arcs. Like in some of the other approaches, arcs are annotated with positive and negative tokens. Moreover, each arc has a context status denoting whether a positive token may still arrive. Using a four-phase approach tokens and statuses are propagated thus yielding a new kind of semantics which overcomes some of the well-known problems related to OR-joins in EPCs.

1 Introduction

The Event-driven Process Chain (EPC) is a business process modeling language for the represention of temporal and logical dependencies of activities in a business process (see [KNS92]). EPCs offer *function type* elements to capture the activities of a process and *event type* elements describing pre- and post-conditions of functions. Some EPC definitions also include *process interface type* elements. A process interface is a syntax element that links two consecutive EPCs: at the end of the first EPC, a process interface points to the second EPC, and the at beginning of the second, there is a process interface representing the preceding EPC. Furthermore, there are three kinds of *connector types* (i.e. AND, OR, and XOR) for the definition of complex routing rules. Connectors have either multiple incoming and one outgoing arc (join connectors) or one incoming and multiple outgoing arcs (split connectors). As a syntax rule, functions and events have to alternate, either directly or indirectly when they are linked via one or more connectors. Moreover, OR- and XOR-splits after events are not allowed, since events cannot make decisions. Control flow arcs are used to link elements.

The informal (or intended) semantics of an EPC can be described as follows. The AND-split activates all subsequent branches in a concurrent fashion. The XOR-split represents a choice between one of alternative branches. The OR-split triggers one, two or up to all of multiple branches based on conditions. In both cases of the XOR- and OR-split, the activation conditions are given in events subsequent to the connector. Accordingly, event-function-splits are forbidden with XOR and OR to avoid the situation where the activation conditions do not become clear in the model. The AND-join waits for all in-

coming branches to complete, then it propagates control to the subsequent EPC element. The XOR-join merges alternative branches. The OR-join synchronizes all active incoming branches, i.e., it needs to know whether the incoming branches may receive tokens in the future. This feature is called non-locality since the state of all (transitive) predecessor nodes has to be considered.

Several formal approaches were presented for the definition of EPC semantics. A particular problem of these semantics is that refining a function of an EPC with a structured OR-block can yield unexpected behavior. We will illustrate this problem by the help of an example. Against this background, we present a concept for the definition of EPC semantics based on state and context. The remainder of the paper is structured as follows. Section 2 provides a definition of EPC syntax. Section 3 discusses problems of existing EPC formalizations and an approach to formalize EPCs based on state and context. Section 4 concludes the paper and gives an outlook on future research.

2 EPC Syntax

There is not only one, but there are several approaches towards the formalization of EPC syntax. A reason for that is that the original EPC paper introduces them only in an informal way (see [KNS92]). The subsequent syntax definition of flat EPCs essentially follows the presentation in [NR02] and [MN03]. If it is clear from the context that a flat EPC is discussed, the term EPC will be used instead for brevity. Please note that an initial marking as proposed in [Rum99, NR02] is not included in the syntax definition, but discussed in the context of soundness in Section 3.6.

Definition 1 (Flat EPC)**.** A flat $EPC = (E, F, P, C, l, A)$ consists of four pairwise disjoint and finite sets E, F, C, P, a mapping $l : C \rightarrow \{and, or, xor\}$, and a binary relation $A \subseteq (E \cup F \cup P \cup C) \times (E \cup F \cup P \cup C)$ such that

- An element of E is called *event*. $E \neq \emptyset$.
- An element of F is called *function*. $F \neq \emptyset$.
- An element of P is called *process interface*.
- An element of C is called *connector*.
- The mapping l specifies the type of a connector $c \in C$ as and, or, or xor.
- A defines the control flow as a coherent, directed graph. An element of A is called an *arc*. An element of the union $N = E \cup F \cup P \cup C$ is called a *node*.

In order to allow for a more concise characterization of EPCs, notations are introduced for preset and postset nodes, incoming and outgoing arcs, paths, transitive closure, corona, and several subsets.

Definition 2 (Preset and Postset of Nodes)**.** Let N be a set of $nodes$ and $A \subseteq N \times N$ a binary relation over N defining the arcs. For each $node\ n \in N$, we define its $preset\ {\bullet}n = \{x \in N | (x, n) \in A\}$, and its $postset\ n{\bullet} = \{x \in N | (n, x) \in A\}$.

Definition 3 (Incoming and Outgoing Arcs)**.** Let N be a set of $nodes$ and $A \subseteq N \times N$ a binary relation over N defining the arcs. For each $node\ n \in N$, we define the set

of incoming arcs $n_{in} = \{(x,n)|x \in N \wedge (x,n) \in A\}$, and the set of outgoing arcs $n_{out} = \{(n,y)|y \in N \wedge (n,y) \in A\}$.

Definition 4 (Paths, Connector Chains, and Transitive Closure)**.** Let $a, b \in N$ be two nodes of an EPC. A *path* $a \hookrightarrow b$ refers to a sequence of nodes $n_1, \ldots, n_k \in N$ with $a = n_1$ and $b = n_k$ such that for all $i \in 1, \ldots, k$ holds: $(n_1, n_2), \ldots, (n_i, n_{i+1}), \ldots, (n_{k-1}, n_k) \in A$. This includes the empty path of length zero, i.e., for any node $a : a \hookrightarrow a$. If $a, b \in N$ and $n_2, \ldots, n_{k-1} \in C$, the path $a \overset{c}{\hookrightarrow} b$ is called *connector chain*. This includes the empty connector chain, i.e., $a \overset{c}{\hookrightarrow} b$ if $(a,b) \in A$. a, b. The *transitive closure* $A*$ contains (n_1, n_2) if there is a non-empty path from n_1 to n_2, i.e., there is a a non-empty set of arcs of A leading from n_1 to n_2. For each *node* $n \in N$, we define its *transitive preset* $*n = \{x \in N|(x,n) \in A*\}$, and its *transitive postset* $n* = \{x \in N|(n,x) \in A*\}$.

Definition 5 (Upper Corona, Lower Corona)**.** For a node $n \in N$ of an EPC, its upper corona is defined as $\overset{c}{*}\mathrm{n}= \{v \in (E \cup F \cup P)|v \overset{c}{\hookrightarrow} n\}$. It includes those non-connector nodes of the transitive preset that reach n via a connector chain. In analogy, its lower corona is defined as $n\overset{c}{*}= \{w \in (E \cup F \cup P)|n \overset{c}{\hookrightarrow} w\}$.

Definition 6 (Subsets)**.** For an EPC, we define the following subsets of its nodes and arcs:

- $E_s = \{e \in E \mid |\bullet e| = 0\}$ being the set of start-events,
  $E_{int} = \{e \in E \mid |\bullet e| = 1 \wedge |e\bullet| = 1\}$ being the set of intermediate-events, and
  $E_e = \{e \in E \mid |e\bullet| = 0\}$ being the set of end-events.
- $P_s = \{p \in P \mid |\bullet p| = 0\}$ being the set of start-process-interfaces and
  $P_e = \{p \in P \mid |p\bullet| = 0\}$ being the set of end-process-interfaces.
- $J = \{c \in C \mid |\bullet c| > 1$ and $|c\bullet| = 1\}$ as the set of join- and
  $S = \{c \in C \mid |\bullet c| = 1$ and $|c\bullet| > 1\}$ as the set of split-connectors.
- $C_{and} = \{c \in C \mid (c, and) \in l\}$ being the set of and-connectors,
  $C_{xor} = \{c \in C \mid (c, xor) \in l\}$ being the set of xor-connectors, and
  $C_{or} = \{c \in C \mid (c, or) \in l\}$ being the set of or-connectors.
- $J_{and} = \{c \in J \mid (c, and) \in l\}$ being the set of and-join-connectors,
  $J_{xor} = \{c \in J \mid (c, xor) \in l\}$ being the set of xor-join-connectors,
  $J_{or} = \{c \in J \mid (c, or) \in l\}$ being the set of or-join-connectors,
- $S_{and} = \{c \in S \mid (c, and) \in l\}$ being the set of and-split-connectors,
  $S_{xor} = \{c \in S \mid (c, xor) \in l\}$ being the set of xor-split-connectors, and
  $S_{or} = \{c \in S \mid (c, or) \in l\}$ being the set of or-split-connectors.
- $C_{EF} = \{c \in C \mid \overset{c}{*}c \subseteq E \wedge c\overset{c}{*} \subseteq (F \cup P)\}$ as the set of event-function-connectors (ef-connectors) and
  $C_{FE} = \{c \in C \mid \overset{c}{*}c \subseteq (F \cup P) \wedge c\overset{c}{*} \subseteq E\}$ as the set of function-event-connectors (fe-connectors).
- $A_{EF} \subseteq (E \cup C_{EF}) \times (F \cup P \cup C_{EF})$ as the set of event-function-arcs and
  $A_{FE} \subseteq (F \cup P \cup C_{FE}) \times (E \cup C_{FE})$ as the set of function-event-arcs.
- $A_s \subseteq \{(x,y) \in A \mid x \in E_s\}$ as the set of start-arcs,
  $A_{int} \subseteq \{(x,y) \in A \mid x \notin E_s \wedge y \notin E_e\}$ as the set of intermediate-arcs, and
  $A_e \subseteq \{(x,y) \in A \mid y \in E_e\}$ as the set of end-arcs.

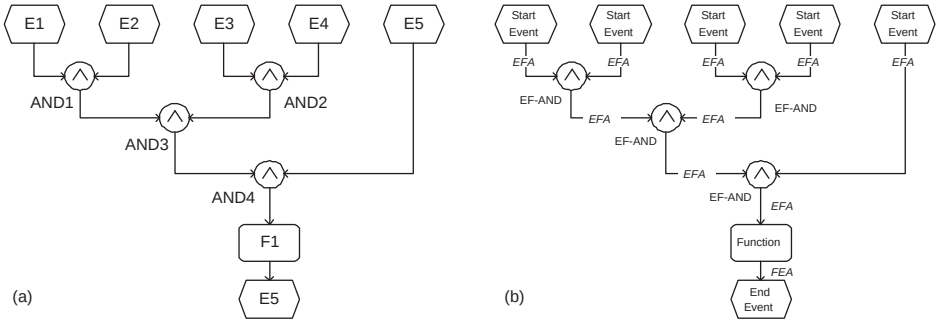


Figure 1: An EPC (a) with labelled nodes and (b) its nodes related to the subsets of Definition 6.

Figure 1 illustrates the different subsets of an EPC. Consider for example the connector $AND3$. It is an event-function-connector since its upper corona, i.e. those non-connector nodes from which there is a connector chain to $AND3$, contains only events and its lower corona contains functions only, in this case exactly one function. Furthermore, the arc from $AND1$ to $AND3$ is an event-function-arc since it connects two event-function-connectors. Note that arcs from events to event-function-connectors and arcs from event-function-connectors to functions are event-function-arcs, too.

In the remainder we assume an EPC to satisfy the following requirements.

Definition 7 (Syntactically Correct EPC)**.** An $EPC = (E, F, P, C, l, A)$ is called syntactically correct, if it fulfills the requirements:

1. EPC is a simple, directed, coherent, and antisymmetric graph.
2. There are no connector cycles, i.e. $\forall a, b \in C$: if $a \neq b$ and $a \overset{c}{\hookrightarrow} b$, then $\nexists b \overset{c}{\hookrightarrow} a$.
3. $|E| \geq 2$. There are at least two events in an EPC.
4. $|F| \geq 1$. There is at least one function in an EPC.
5. Events have at most one incoming and one outgoing arc.
   $\forall e \in E : |\bullet e| \leq 1 \wedge |e\bullet| \leq 1$.
6. Functions have exactly one incoming and one outgoing arcs.
   $\forall f \in F : |\bullet f| = 1 \wedge |f\bullet| = 1$.
7. Process interfaces have one incoming or one outgoing arcs.
   $\forall p \in P : (|\bullet p| = 1 \wedge |p\bullet| = 0) \vee (|\bullet p| = 0 \wedge |p\bullet| = 1)$.
8. Connectors have one incoming and multiple outgoing arcs or multiple incoming and one outgoing arc. $\forall c \in C : (|\bullet c| = 1 \wedge |c\bullet| > 1) \vee (|\bullet c| > 1 \wedge |c\bullet| = 1)$.
9. Events must have function, process interface, and fe-connector nodes in the preset, and function, process interface and ef-connector nodes in the postset.
   $\forall e \in E : \bullet e \subseteq (F \cup P \cup C_{FE}) \wedge e\bullet \subseteq (F \cup P \cup C_{EF})$.
10. Functions must have events and ef-connectors in the preset and events and fe-connectors in the postset.
    $\forall f \in F : \bullet f \subseteq (E \cup C_{EF}) \wedge f\bullet \subseteq (E \cup C_{FE})$.
11. Process interfaces are connected to events only.
    $\forall p \in P : \bullet p \subseteq E \wedge p\bullet \subseteq E$.

12. Connectors must have either functions, process interfaces, and fe-connectors in the preset and functions, process interfaces, and ef-connectors in the postset or events and ef-connectors in the preset and events and fe-connectors in the postset.
$\forall c \in C : (\bullet c \subseteq (F \cup P \cup C_{FE})) \wedge c\bullet \subseteq (F \cup P \cup C_{EF}) \vee$
$(\bullet c \subseteq (E \cup C_{EF}) \wedge c\bullet \subseteq (E \cup C_{FE}))$.
13. Arcs either connect events and ef-connectors with functions, process interfaces, and ef-connectors or functions, process interfaces, and fe-connectors with events and fe-connectors.
$\forall a \in A : (a \in (E \cup C_{EF}) \times (F \cup P \cup C_{EF})) \vee (a \in (F \cup P \cup C_{FE}) \times (E \cup C_{FE}))$.

3 EPC Semantics

In addition to related work on the syntax of EPCs, there are several contributions towards the formalization of EPC semantics. This section first illustrates the semantical problems related to the OR join using an illustrative set of examples, then it gives a historical overview of semantical definitions described in literature, and provides a formalization for EPCs that is used throughout this thesis.

3.1 Informal Semantics as a Starting Point

Before discussing EPC formalization problems, we need to establish an informal understanding of state representation and state changes of an EPCs. Although we provide a formal definition not before Section 3.4, the informal declaration of state concepts helps to discuss formalization issues in this section. The *state*, or *marking*, of an EPC is defined by assigning a number of *tokens* (or process folders) to its arcs.[1] The formal semantics of an EPC define which state changes are possible for a given marking. These state changes are formalized by a *transition relation*. A node is called *enabled* if there are enough tokens on its incoming arcs that it can fire, i.e., a state change defined by a transition can be applied. This process is also called *firing*. A firing of a node n consumes tokens from its input arcs n_{in} and produces tokens at its output arcs n_{out}. The formalization of whether an OR-join is enabled is a non-trivial issue since not only the incoming arcs have to be considered. The sequence $\tau = n_1 n_2 ... n_m$ is called a firing sequence if it is possible to execute a sequence of steps, i.e., after firing n_1 it is possible to fire n_2, etc. Through a sequence of firings the EPC moves from one *reachable* state to the next. The *reachability graph* of an EPC represents how states can be reached from each other. A state that is not a final state, but from which no other state can be reached is called a *deadlock*. The notion of a final state will be formally defined in Section 3.6.

[1] This state representation based on arcs reflects the formalization of *Kindler* [Kin06] and can be related to arcs between tasks in YAWL that are interpreted as implicit conditions [AH05]. Other approaches assign tokens to the nodes of an EPC, e.g., [Rum99].

3.2 EPC Formalization Problems

We have briefly stated that the OR-join synchronizes all active incoming branches. This bears a non-trivial problem: if there is a token on one incoming arc, does the OR-join have to wait or not? Following the informal semantics of EPCs, it is only allowed to fire if it is not possible that a token might arrive at the other incoming arcs (see [NR02]). In the following subsection, we will show what the formal implications of these intended semantics are. Before that, we introduce some example EPCs. The discussion of them raises some questions that are not answered yet. Instead, we revisit them afterwards to illustrate the characteristics of the different formalization approaches.

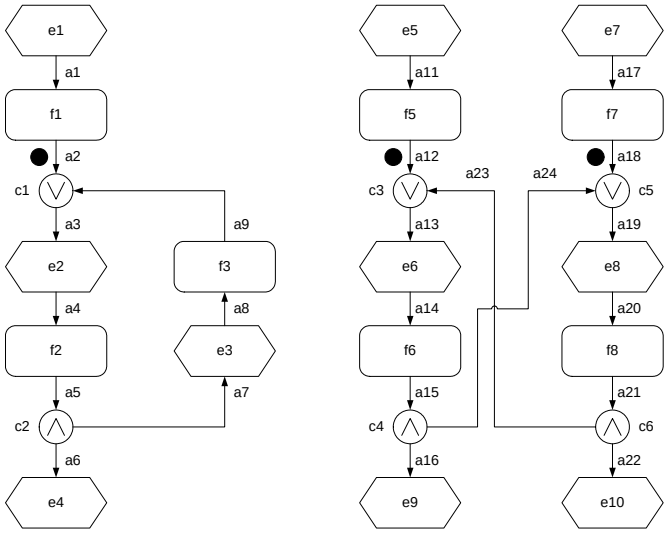


Figure 2: EPCs (a) with one OR-join and (b) with two OR-joins on the loop

Figure 2(a) shows an EPC with an OR-join on a loop. There is a token on arc $a2$ from function $f1$ to the OR-join $c1$. The question is whether $c1$ can fire. If it could fire, then it would be possible that a token may arrive at arc $a9$ from $f3$ to the join. This would imply that it should wait and not fire. On the other hand, if it must wait, it is not possible that a token might arrive at $a9$. Figure 2(b) depicts an EPC with two OR-joins, $c3$ and $c5$, on a loop which are both enabled (cf. [ADK02]). Here, the question is whether both can fire or none of them. Since the situation is symmetric it seems not reasonable that only one should be allowed to fire.

The situation might be even more complicated as Figure 3 illustrates (cf. [Kin06]). This EPC includes a loop with three OR-joins: $c1$, $c3$, and $c5$. All of them are enabled. Following the informal semantics the first OR-join $c1$ is allowed to fire if it is not possible for a token to arrive on arc $a21$ from the AND-split $c6$. To put it differently, if $c1$ is allowed to fire, it is possible for a token to arrive on arc $a7$ that leads to the OR-join $c3$. Furthermore,

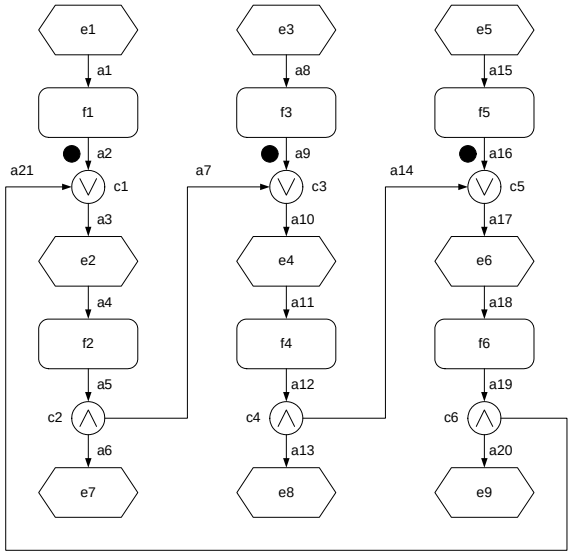


Figure 3: EPCs with three OR-joins on the loop

the OR-join $c5$ could eventually fire. Finally, the first OR-join $c1$ would have to wait for that token before firing. In short, if $c1$ could fire, it would have to wait. One can show that this holds also the other way around: if it could not fire, it would not have to wait. Furthermore, this observation also holds for the two other OR-joins. In the subsequent section, we will discuss whether this problem can be resolved.

Refinement is another issue related to OR-joins. Figure 4 shows two versions of an EPC process model. In Figure 4(a) there is a token on $a7$. The subsequent OR-join $c2$ must wait for this token and synchronize it with the second token on $a5$ before firing. In Figure 4(b) the sequence $e3$-$a7$-$f3$ is refined with a block of two branches between an OR-split $c3a$ and an OR-join $c3b$. The OR-join $c2$ is enabled and should wait for the token on $a7f$. The question here is whether such a refinement might change the behavior of an OR-join. Figure 4 is just one simple example. The answer to this question may be less obvious if the refinement is introduced in a loop that already contains an OR-join. Figure 5 shows a respective case of an OR-join $c1$ on a loop that is refined with an OR-Block $c3a$-$c3b$. One would expect that the EPC of Figure 4(a) exhibits the same behavior as the one in (b). In the following subsection, we will discuss these questions from the perspective of different formalization approaches.

3.3 Approaches to EPC Semantics Formalization

The transformation to Petri nets plays an important role in early formalizations of EPC semantics. In *Chen and Scheer* [CS94] the authors define a mapping to colored Petri nets

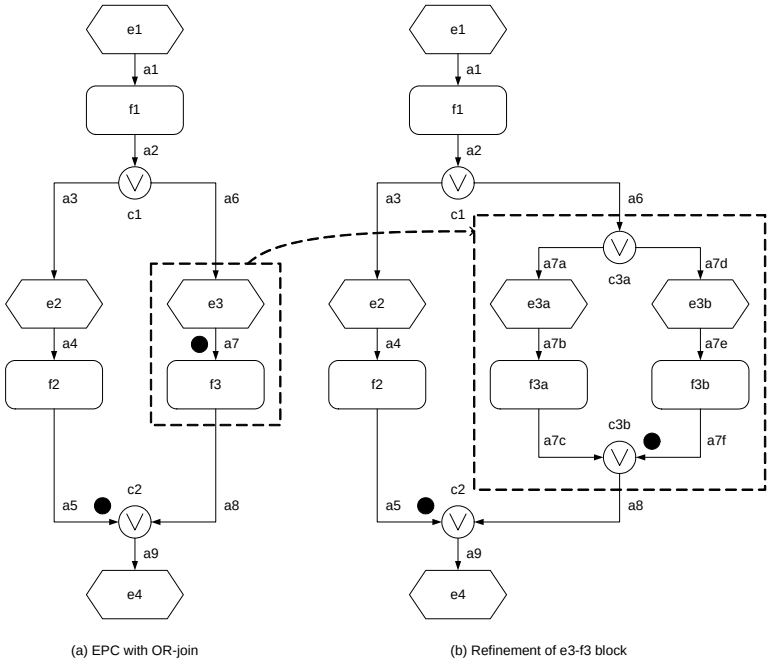


Figure 4: EPC refined with an OR-Block

and address the non-local synchronization behavior of OR-joins. This formalization builds on the assumption that an OR-split always matches a corresponding OR-join. The colored token that is propagated from the OR-split to the corresponding OR-join signals which combination of branches is enabled. Furthermore, the authors describe the state space of some example EPCs by giving reachability graphs. Yet, this first Petri net semantics for EPCs has mainly two weaknesses. First, a formal algorithm to calculate the state space is missing. Second, the approach is restricted to EPCs with matching OR-split and -join pairs. Therefore, this approach does not provide semantics for the EPCs shown in figures 2 and 3. Even though the approach is not formalized in all its details, it should be able to handle the refined EPC of Figure 4(b) and the inner OR-join $c3b$ in Figure 4(b).

The transformation approach by *Langner, Schneider, and Wehler* [LSW98] maps EPCs to Boolean nets in order to define formal semantics. Boolean nets are a variant of colored Petri nets whose token colors are 0 (negative token) and 1 (positive token). Connectors propagate both negative and positive tokens according to its logical type. This mechanism is able to capture the non-local synchronization semantics of the OR-join similar to dead-path elimination in workflow systems (see [LA94]). The XOR-join only fires if there is one positive token on incoming branches and a negative token on all other incoming branches. Otherwise it blocks. A drawback of this semantics definition is that the EPC syntax has to be restricted: arbitrary structures are not allowed. If there is a loop it must have an XOR-join as entry point and an XOR-split as exit point. This pair of connectors in a cyclic structure is mapped to one place in the resulting Boolean net. As a consequence,

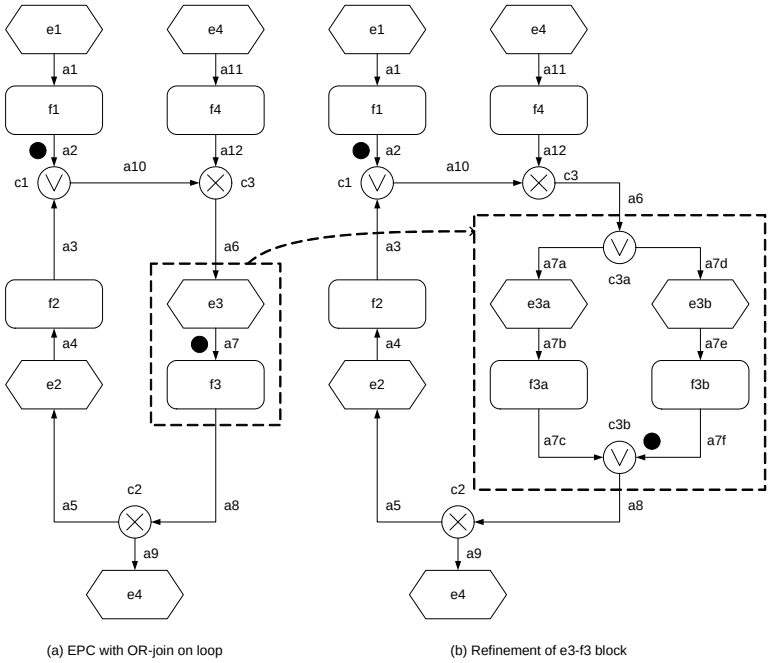


Figure 5: Cyclic EPC refined with an OR-Block

this approach does not provide semantics for the EPCs in Figures 2 and 3. Yet, it is able to deal with any kind of OR-join that is not an entry or an exit to a loop. Accordingly, the Boolean nets define the expected semantics of the refined EPC of Figure 4(b) and of the inner OR-Block introduced as a refinement in Figure 4(b).

Van der Aalst [Aal99] presents an approach to derive Petri nets from EPCs, but he does not give a mapping for OR-connectors because of the semantical problems illustrated in Section 3.2. This mapping provides clear semantics for XOR- and AND-connectors as well as for the OR-split, but since the OR-join is not formalized the approach does not provide semantics for the EPCs of Figures 2 to 5. Dehnert presents an extension of this approach by mapping the OR-join to a Petri net block [DA04]. Since the resulting Petri net block may or may not necessarily synchronize multiple tokens at runtime (i.e., a non-deterministic choice), its state space is larger than the actual state space with synchronization. Based on the so-called relaxed soundness criterion it is possible to check whether a join should synchronize (cf. [DA04]).

In [Rit00] *Rittgen* discusses the OR-join. He proposes to distinguish three types of OR-joins already on the syntactic level: every-time, first-come, and wait-for-all. The every-time OR-join basically reflects XOR-join behavior; the first-come OR-join passes the first incoming token and blocks afterwards; and the wait-for-all OR-join depends on a matching split similar to the approach of *Chen and Scheer*. This proposal could provide a semantics for the example EPCs of Figures 2 to 5. If we assume an every-time semantics, all OR-joins of the example EPCs could fire. While the loops would not block in this case, there

would be no synchronization at all which contradicts the intended OR-join semantics. If the OR-joins behave according to the first-come semantics, all OR-joins could fire. Yet, there would also be no synchronization and the loops could be run only once. If the OR-joins had wait-for-all semantics, we would have the same problems as before with the loops. Altogether, the proposal by Rittgen does not provide a satisfactory solution to the formalization problem.

Nüttgens and Rump [NR02] define a transition relation for EPCs that addresses also the non-local semantics of the OR-join, yet with a problem: the transition relation for the OR-join refers to itself under negation. *Van der Aalst, Desel, and Kindler* show, that a fixed point for this transition relation does not always exist [ADK02]. They present an example to prove the opposite: an EPC with two OR-joins on a circle that wait for each other as depicted in Figure 2(b). This vicious circle is the starting point for the work of *Kindler* towards a sound mathematical framework for the definition of non-local semantics for EPCs. In a series of papers [Kin06], *Kindler* elaborates on this problem in detail. The technical problem is that for the OR-join transition relation R depends upon R itself in negation. Instead of defining one transition relation, he considers a pair of transition relations (P, Q) on the state space Σ of an EPC and a monotonously decreasing function $R : 2^{\Sigma \times N \times \Sigma} \rightarrow 2^{\Sigma \times N \times \Sigma}$. Then, a function $\varphi((P, Q)) = (R(Q), R(P))$ has a least fixed point and a greatest fixed point. P is called pessimistic transition relation and Q optimistic transition relation. An EPC is called *clean*, if $P = Q$. For most EPCs, this is the case. Some EPCs such as the vicious circle EPC are *unclean* since the pessimistic and the optimistic semantics do not coincide. Moreover, *Cuntz* provides an example of a clean EPC which is refined with another clean EPC and becomes unclean [Cun04, p.45]. *Kindler* also shows that there are even acyclic EPCs that are unclean (see [Kin06, p.38]). Furthermore, *Cuntz and Kindler* present optimizations for an efficient calculation of the state space of an EPC and a respective prototype implementation called EPC Tools [CK05]. EPC Tools also offers a precise answer to the questions about the example EPCs of Figures 2 to 5.

- Figure 2(a): For the EPC with one OR-join on a loop, there is a fixed point and the connector is allowed to fire.
- Figure 2(b): The EPC with two OR-joins on a loop is not clean. Therefore, it is not clear whether the optimistic or the pessimistic semantics should be considered.
- Figure 3: The EPC with three OR-joins is also not clean, i.e., the pessimistic deviates from the optimistic semantics.
- Figure 4(a): The OR-join $c2$ must wait for the second token on $a7$.
- Figure 4(b): The OR-join $c2$ must wait for the second token on $a7f$.
- Figure 5(a): The OR-join $c1$ must wait for the second token on $a7$.
- Figure 5(b): The OR-join $c1$ is allowed to fire, the second OR-join $c2$ in the OR-block must wait.

Even though the approach by *Kindler* provides semantics for a large subclass of EPCs, i.e. clean EPCs, there are some cases like the EPCs of Figure 2(b) and 3 that do not have semantics. The theorem by *Kindler* proves that it is not possible to give these EPCs semantics as long as the transition relation is defined with a self-reference under negation. Furthermore, such a semantics definition may imply some unexpected results, e.g. if an

EPC such as that of Figure 5(a) behaves differently than its refinement as given in Figure 5(b).

Van der Aalst and Ter Hofstede defined a workflow language called YAWL [AH05] which also offers an OR-join with non-local semantics. As *Mendling, Moser, and Neumann* propose a transformation semantics for EPCs based on YAWL [MMN06], we discuss in the following paragraph how the OR-join behavior is formalized in YAWL. In [AH05], the authors propose a definition of the transition relation $R(P)$ with a reference to a second transition relation P that ignores all OR-joins. A similar semantics that is calculated on history-logs of the process is proposed by *Van Hee, Oanea, Serebrenik, Sidorova, and Voorhoeve* in [HOS$^+$06]. The consequence of this definition can be illustrated using the example EPCs.

- Figure 2(a): The single OR-join on the loop can fire.
- Figure 2(b): The two OR-joins on the loop can fire.
- Figure 3: The three OR-joins on the loop can fire.
- Figure 4(a): The OR-join $c2$ must wait for the second token between e3 and f3.
- Figure 4(b): Both OR-joins can fire.
- Figure 5(a): The OR-join $c1$ must wait for the second token between e3 and f3.
- Figure 5(b): Both OR-joins can fire.

Kindler criticizes that each choice for defining P "appears to be arbitrary or ad hoc in some way" [Kin06] and uses the pair (P, Q) instead. The example EPCs illustrate that the original YAWL semantics provide for a limited degree of synchronization. Consider for example the vicious circle EPC with three OR-joins: all are allowed to fire, but if one does so, the subsequent OR-join has to wait. Furthermore, the refined EPCs exhibit different behavior than their unrefined counterparts since OR-joins are ignored, i.e., they are considered to be not able to fire.

Wynn, Edmond, van der Aalst, and ter Hofstede illustrate that these OR-join semantics in YAWL exhibit some non-intuitive behavior when OR-join depend upon each other [WEAH05]. Therefore, they present a novel approach based on a mapping to Reset nets. If an OR-join can fire (i.e. $R(P)$) is decided depending on (a) a corresponding Reset net (i.e. P) that treats all OR-joins as XOR-joins[2] and (b) a predicate called *superM* that hinders firing if an OR-join is on a directed path from another enabled OR-join. In particular, the Reset net is evaluated using backward search techniques that grant coverability to be decidable (see e.g. [FS01]). A respective verification approach for YAWL nets is presented in [WAHE06]. Using these semantics, the example EPCs behave as follows:

- Figure 2(a): The single OR-join on the loop can fire since *superM* evaluates to false and hence no more tokens can arrive at c_1.
- Figure 2(b): The two OR-joins are not enabled since *superM* evaluates to true because if the respectively other OR-join is replaced by an XOR-join an additional token may arrive.

[2] In fact, [WEAH05] proposes two alternative treatments for the "other OR-joins" when evaluating an OR-joins: treat them either as XOR-joins (optimistic) or as AND-joins (pessimistic). However, the authors select the optimistic variant because the XOR-join treatment of other OR-joins matches more closely the informal semantics of the OR-join.

- Figure 3: The three OR-joins are not enabled because if one OR-join assumes the other two to be XOR-joins then this OR-join has to wait.
- Figure 4(a): The OR-join $c2$ must wait for the second token on *a7*.
- Figure 4(b): The OR-join $c2$ must wait for the second token on *a7f*.
- Figure 5(a): The OR-join *c1* must wait for the token on *a7*.
- Figure 5(b): The OR-join *c1* must wait because if $c3b$ is assumed to be an XOR-join a token may arrive via $a3$. The OR-join *c3b* must also wait because if $c1$ is an XOR-join another token may move to $a7c$.

The novel approach based on Reset nets provides interesting semantics but in some cases also leads to deadlocks.

OR-join semantics	Limitations
[CS94]	OR-join must match OR-split
[LSW98]	Joins as loop entry undefined
[Rit00] every-time	missing synchronization
[Rit00] first-come	OR-join can block
[Rit00] wait-for-all	OR-join as loop entry undefined
[Kin06]	EPC can be unclean
[AH05]	limited synchronization
[WAHE06]	OR-join may block

Table 1: Overview of EPC semantics and their limitations

Table 1 summarizes existing work on the formalization of the OR-join. Several early approaches define syntactical restrictions such as OR-splits to match corresponding OR-joins or models to be acyclic (see [CS94, LSW98, Rit00]). Newer approaches impose little or even no restrictions (see [Kin06, AH05, WAHE06]), but exhibit unexpected behavior for OR-block refinements on loops with further OR-joins on it. The solution based on Reset nets seems to be most promising from the intuition of its behavior. Yet, it requires extensive calculation effort since it depends upon backward search to decide coverability (Note that reachability is undecidable for reset nets illustrating the computational complexity of the OR-join in the presence of advanced routing constructs). In the following subsection, we propose a novel approach that addresses the refinement problems of the Reset nets semantics and that provides a more efficient solution since all OR-join decisions can be taken with local knowledge.

3.4 A Novel Approach towards EPC Semantics

In this subsection, we introduce a novel formalization of the EPC semantics. The principal idea of these semantics lends some concepts from *Langner, Schneider, and Wehler* [LSW98] and adapts the idea of Boolean nets with true and false tokens in an appropriate manner. Furthermore, we utilize the notations of *Kindler* [Kin06] whenever possible and modify them where needed. The transition relation that we will formalize afterwards depends on the state and the context of an EPC. The *state* of an EPC is basically an assign-

ment of positive and negative tokens to the arcs. Positive tokens signal which functions have to be carried out in the process, negative tokens indicate which functions are to be ignored. The transition rules of AND- and OR-connectors are adopted from the Boolean nets formalization which facilitates synchronization of OR-joins in structured blocks. In order to allow for a more flexible utilization of XOR-connectors in cyclic structure, we modify and extend the approach of Boolean nets in three ways:

1. XOR-splits produce positive tokens on their output arcs, but no negative tokens. XOR-joins fire each time there is a positive token on an incoming arc. This mechanism provides the expected behavior in both structured XOR-loops and structured XOR-blocks.
2. In order to signal OR-joins that it is not possible to have a positive token on an incoming branch, we define the *context* of an EPC. The context assigns a status of *wait* or *dead* to each arc of an EPC. A wait context indicates that it is still possible that a positive token might arrive; a dead context status means that no positive token can arrive anymore. For example, XOR-splits produce a dead context on those output branches that are not taken and a wait context on the output branch that receives a positive token. A dead context at an input arc is then used by an OR-join to determine whether it has to synchronize with further positive tokens or not.
3. The propagation of context status and state tokens is arranged in a four phase cycle: (a) dead context, (b) wait context, (c) negative token, and (d) positive token propagation.
   (a) In this phase, all *dead context* information is propagated in the EPC until no new dead context can be derived.
   (b) Then, all *wait context* information is propagated until no new wait context can be derived. It is necessary to have two phases (i.e., first the dead context propagation and then the wait context propagation) in order to avoid infinite cycles of context changes (details below).
   (c) After that, all *negative tokens* are propagated until no negative token can be propagated anymore. This phase can neither run into an endless loop (details below).
   (d) Finally, one of the enabled nodes is selected and propagates *positive tokens* leading to a new iteration of the four phase cycle.

In the following subsection, we first give an example to illustrate the behavior of the EPC semantics before defining state, context, and the transition relation. Then, we define the initialization of an EPC. After that, we present the transition relations for the two phases of dead context and wait context propagation. Then, we discuss the termination of these phases and why a separation in dead context and wait context propagation is necessary. Subsequently, we define the transition relation for the phase of negative token propagation and its termination. Finally, the positive token propagation is presented.

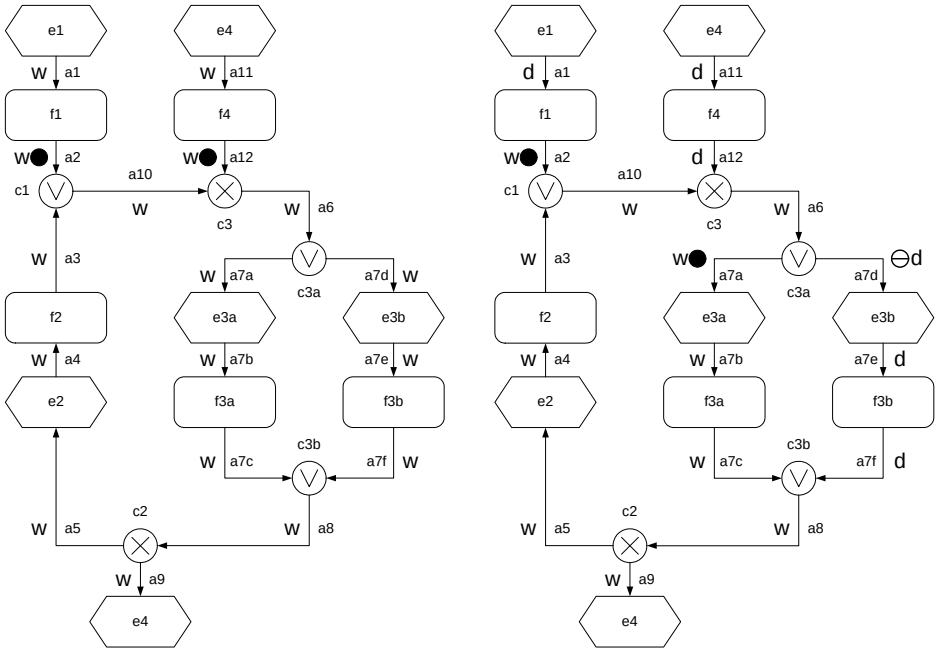
Revisiting the cyclic EPC refined with an OR-block Figure 6 revisits the example of the cyclic EPC refined with an OR-block that we introduced as Figure 5 in Section 3.2.

In Figure 6(a) there are two positive tokens on the arcs $a2$ and $a12$. The context status is indicated by a letter next to the arc: w for wait and d for dead. In (a) all arcs are in a wait context status which implies that the OR-join $c1$ is not allowed to fire, but has to synchronize with positive and negative tokens that might arrive on arc $a3$. The XOR-join is allowed to fire without considering the second arc $a10$. In (b) the OR-split $c3a$ has fired (following the execution of c3) and happened to produce a positive token on $a7a$ and a negative token on $a7d$. Accordingly, the context of $a7d$ is changed to dead. This dead context is propagated down to arc $a7f$. The rest of the context remains unchanged. The state shown in (b) is followed by (c) where the positive and the negative tokens are synchronized at the connector $c3b$ and one positive token is produced on the output arc $a8$. Please note that the OR-join $c3b$ does not synchronize with the other OR-join $c1$ that is also on the loop. In the Kindler and the Reset nets semantics, $c3b$ would have to wait for the token from $a2$. Here, the wait context propagation is blocked by the negative token. In (d) the XOR-split $c2$ produces a positive token on $a9$ and a dead context on $a5$. This dead context is propagated to $a3$ in the dead context propagation phase, but not further. In the wait context propagation phase, this dead context is changed to wait which is propagated from $c2$. As a consequence, the OR-join $c1$ is not enabled. This example permits two observations. First, the context propagation blocks OR-joins that are entry points to a loop in a wait position since the self-reference is not resolved. Second, the XOR-split produces a dead context, but not a negative token. The disadvantage of producing negative tokens would be that the EPC was flooded with negative tokens if an XOR-split was used as an exit of a loop. These tokens would give downstream joins the wrong information about the state of the loop since it would be still live. An OR-join could then synchronize with a negative token while a positive token was still in the loop. In contrast to that, the XOR-split as a loop exit produces a dead context. Since there is a positive token in the loop, it is overwritten in the wait context propagation phase. Downstream OR-joins then have the correct information that there are still tokens to wait for.

Definition of State and Context We define both state and context as an assignment to the arcs. The EPC transition relation defines which state and/or context changes are allowed for a given state/context. We will first illustrate this before defining state and context formally. As we define the EPC semantics as a transition system based on state and context, we refer to it as a state-context-system.

Definition 8 (State and Context)**.** For an $EPC = (E, F, C, l, A)$ the mapping $\sigma : A \rightarrow \{-1, 0, +1\}$ is called a state (or marking) of an EPC. The positive token captures the state as it is observed from outside the process. It is represented by a black circle. The negative token depicted by a white circle with a minus on it has a similar semantics as the negative token in the Boolean nets formalization. Arcs with no state tokens on them have no circle depicted. Furthermore, the mapping $\kappa : A \rightarrow \{wait, dead\}$ is called a context of an EPC. A wait context is represented by a w and a dead context by a d next to the arc.

Initial and Final State The initial state is the starting point for applying an iteration of the four phase cycle. In [Rum99] the initial state of an EPC is specified as a marking that assigns tokens to one, some, or all start events. While such a definition contains enough



(a) Entering the loop

(b) Entering the OR-block

(c) Synchronizing

(d) Exiting the loop

Figure 6: Example of EPC state changes

information for verification purposes, e.g. by the bundling of start and end events with OR-connectors as proposed in [MMN06], it does not provide executable semantics according to the original definition of EPCs. As pointed out in [Rit00], it is not possible to equate the triggering of a single start event with the instantiation of a new process. This is because EPC start events do not only capture alternative instantiation, but also external events that influence the execution of a running EPC (cf. [CS94]). In our approach, we assume that state and context is known for each of the start arcs. A respective formalization of initial and final state is given in Definitions 11 and 12. In the following, we describe the transition relations of each node $n \in E \cup F \cup C$ in the phases of dead context, wait context, negative and positive token propagation.

Phase 1: Transition Relation for Dead Context Propagation The transition relation for dead context propagation defines rules for deriving a dead context if one or more input arcs of a node have a dead context status. Figure 7 gives an illustration of the transition relation. *Please note that the figure does not depict the fact that the the rules for dead context propagation can only be applied if the respective output arc does not hold a positive or a negative token.* Concrete tokens override context information, e.g., an arc with a positive token will always have a wait context. Rules (a) and (b) indicate that if an input arc of a function or an event is dead, then also the output arc has to have a dead context status. Rule (c) represents that each split-connector propagates a dead context to its output arcs. These transition relations formalize the observation that if an input arc cannot be reached anymore, also its output arcs cannot be reached anymore (unless they already hold positive of negative tokens). The AND-join requires only one dead context status at its input arcs to replicate it at its output arc, see (d). XOR- and OR-joins propagate a dead context if all input arcs are dead, see (e) and (f). It is important to note that a dead context is propagated until it reaches a node where one of the output arcs holds a token or an (X)OR-join where one of the inputs has a wait context.

Phase 2: Transition Relation for Wait Context Propagation The transition relation for wait context propagation defines rules for deriving a wait context if one or more input arcs of a node have a wait context status. Figure 8 gives an illustration of the transition relation. *All transitions can only be applied if the respective output arc does not hold a positive or a negative token.* Rules (a) and (b) show that if an input arc of a function or an event has a wait context, then also the output arc has to have a wait context status. Rule (c) represents that each split-connector propagates a wait context to its output arcs. The AND-join requires all inputs to have a wait context status in order to reproduce it at its output arc, see (d). XOR- and OR-joins propagate a wait context if one of their input arcs has a wait context, see (e) and (f). Similar to the dead context propagation, the wait context is propagated until it reaches a node where one of the output arcs holds a token or an AND-join where one of the inputs has a dead context.

Observations on Context Propagation The transition relations of context propagation permit the following observations:

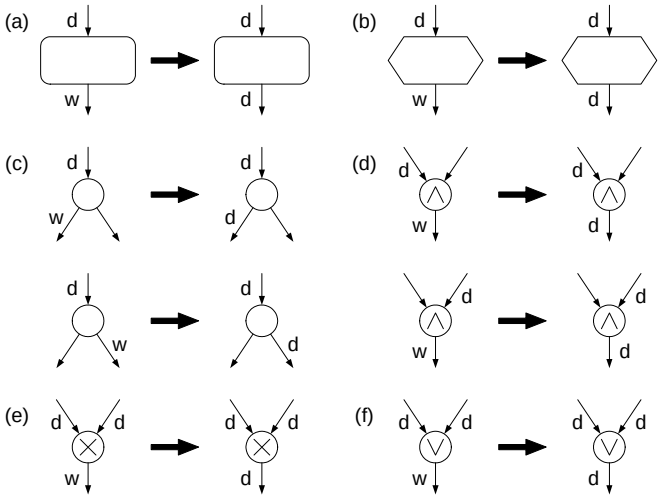


Figure 7: Transition relation for dead context propagation

- *Context changes terminate:* It is intuitive that context propagation cannot run into an infinite loop. It is easy to verify that the first two phases indeed stop. The propagation of dead context stops because the number of arcs is finite, i.e., the number of arcs is an upper bound for the number of times the rules in Figure 7 can be applied. A similar argument applies to the propagation of wait context. As a consequence, the context change phase will always terminate and enable the consideration of new state changes in the subsequent phase.
- *State tokens block context propagation:* The transition relations for context propagation require that the output arcs to be changed do not hold any state token, i.e., arcs with a positive token always have a wait context and arcs with a negative token always have a dead context.
- *Context propagating elements:* Functions, events, and split nodes reproduce the context that they receive at their input arcs.
- *OR- and XOR-joins:* Both these connectors produce a $wait$ context if at least one of the input arcs has a $wait$ context. A $dead$ context is produced if all inputs are $dead$.
- *AND-joins:* AND-joins produce $wait$ context status only if all inputs are $wait$. Otherwise, the output context is set to $dead$.

Figure 9 illustrates the need to perform context propagation in two separate phases and not together in one phase. If there are context changes (a) at $i1$ to and $i2$ the current context enables the firing of the transition rules for both connectors producing a $dead$ context status in $a1$ and a $wait$ context status in $a3$. This leads to a new context in (b) with an additional $dead$ context status in $a2$ and a new $wait$ context status in $a4$. Since both arcs from outside the loop to the connectors are marked in such a way that incoming context changes on the other arc is simply propagated, there is a new context in (c) with a $wait$ status in $a1$ and a $dead$ context status in $a3$. Note that this new context can be propagated

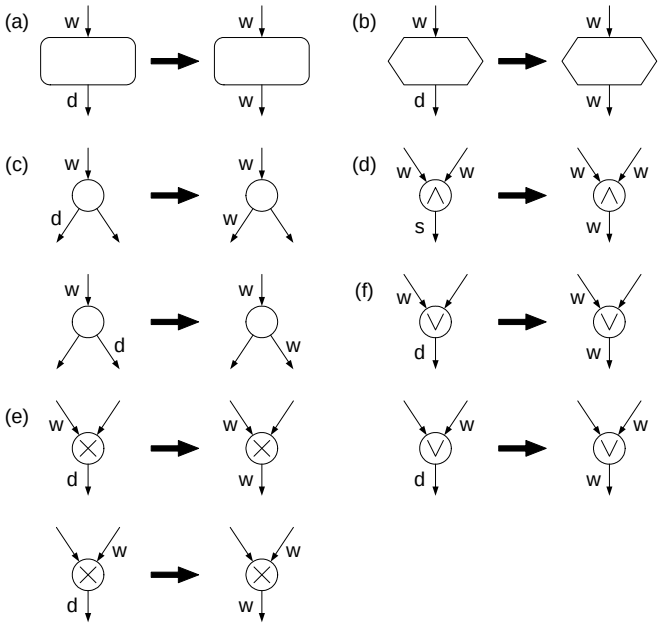


Figure 8: Transition relation for wait context propagation

and this way the initial situation is reached. This can be repeated again and again. Without a sequence of two phases the transitions may continue infinitely and the result may be non-defined.

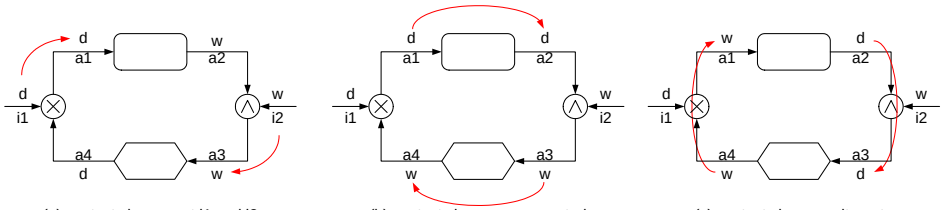


Figure 9: Situation of unstable context changes without two phases

Phase 3: Transition Relation for Negative Token Propagation Negative tokens can result from branches that are not executed after OR-joins or start events. The transition relation for negative token propagation includes four firing rules that consume and produce negative tokens. Furthermore, the output arcs are set to a dead context. Figure 10 gives an illustration of the transition relation. *All transitions can only be applied if all input arcs hold negative tokens and if there is no positive token on the output arc.*

The propagation of negative tokens for an EPC terminates. First, we have to note that the

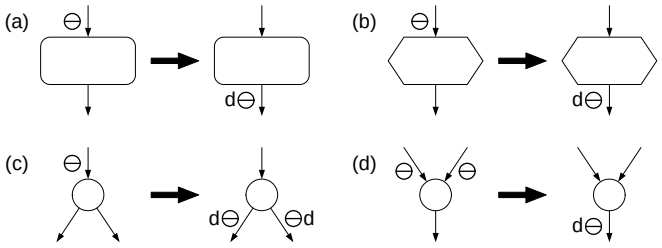


Figure 10: Transition Relation for Negative Token Propagation

number of negative and positive tokens on an arc is limited to one. It is a prerequisite for an infinite propagation that there is a cyclic structure in the process in which the negative token runs into an infinite loop. To this loop there must be an entry point, i.e. a join connector that has an input arc from outside the cycle. According to the transition relation of negative tokens, this join can only produce a negative token on its output arc if it has negative tokens on all inputs. Since there is one arc from an acyclic structure, there is a finite number a of arcs from one or multiple start arcs to this arc limiting the number of negative tokens that can arrive to the number of arcs leading to the join. Therefore, the negative token in the cyclic structure cannot pass the join more than a times.

Phase 4: Transition Relation for Positive Token Propagation The transition relation for positive token propagation specifies firing rules that consume negative and positive tokens from the input arcs of a node to produce positive tokens on its output arcs. Figure 11 gives an respective illustration. Rules (a) and (b) show that functions and events consume positive tokens from the input arc and propagate them to the output arc. Furthermore, and this holds for all rules, consuming a positive token from an arc implies setting this arc to a dead context status. Rules (c) and (d) illustrate that AND-splits consume one positive token and produce one on each output arc while AND-joins synchronize positive tokens on all input arcs to produce one on the output arc. Rule (e) depict the fact that XOR-splits forward positive tokens to one of their output arcs. In contrast to the Boolean net formalization, they do not produce negative tokens, but a dead context on the output arcs that do not receive the token. Correspondingly, XOR-joins (f) propagate each incoming positive token to the output arc, no matter what the context or the state of the other input arcs is. The OR-split (g) produces positive tokens on those output arcs that have to be executed and negative tokens on those that are ignored. Note that the OR-join is the only construct that may introduce negative tokens (apart from start events without an initial token). Rule (h) shows that on OR-join can only fire either if it has full information about the state of its input arcs, i.e., each input has a positive or a negative token, or all arcs that do not hold a token are in a dead context. Finally, each output arc that receives a negative token is set to a dead context and each that gets a positive token is set to a wait context.

This semantics definition based on state and context results in the following behavior for the examples of Section 3.

- Figure 2(a): The single OR-join on the loop produces a wait context at $a9$. There-

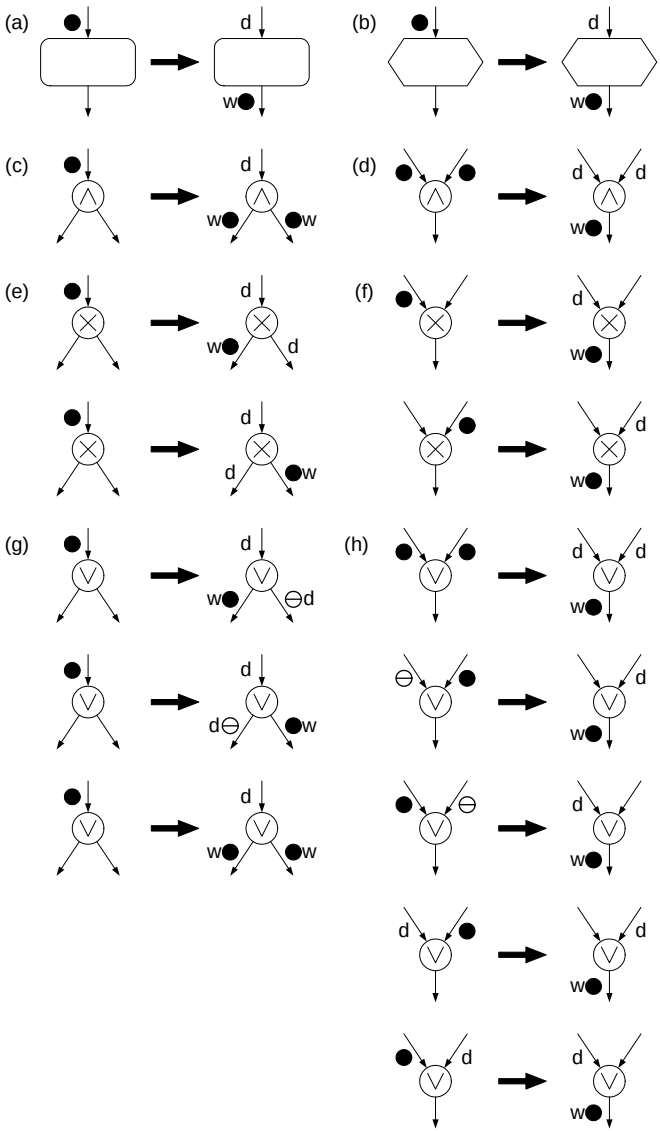


Figure 11: Transition Relation for Positive Token Propagation

fore, it is blocked.
- Figure 2(b): The two OR-joins produce a wait context at $a23$ and $a24$. Therefore, they are both blocked.
- Figure 3: The three OR-joins are blocked due to a wait context at $a7$, $a14$, and $a21$.
- Figure 4(a): The OR-join $c2$ must wait for the second token on *a7*.
- Figure 4(b): The OR-join $c2$ must wait for the second token on *a7f*.
- Figure 5(a): The OR-join *c1* must wait for the token on *a7*.
- Figure 5(b): The OR-join *c1* must wait for the token on *a7*. The OR-split $c3a$ produces a negative token on $a7c$ such that $c3b$ can fire.

3.5 Transition Relation of EPCs

We define the transition relation of EPCs based on state and context mappings σ and κ. In contrast to Petri nets we distinguish the terms marking and state: the term marking refers to state and context collectively.

Definition 9 (Marking of an EPC)**.** For an EPC the mapping $m : A \rightarrow \{-1, 0, +1\} \times \{wait, dead\}$ is called a marking. The set of all markings M of an EPC is called marking space with $M \subseteq A \times \{-1, 0, +1\} \times \{wait, dead\}$. The projection of a given marking m to a subset of arcs $S \subseteq A$ is referred to as m_S. The marking m_a of an arc a can be written as $m_a = (\kappa(a) + \sigma(a)) \cdot a$.

The marking of the arcs $a_1, a_2, a_3 \in A$ can then be written as e.g. $(d-1) \cdot a_1 + w \cdot a_2 + (w+1) \cdot a_3$. Furthermore, we define the transition relation, the initial marking, and the final marking of an EPC.

Definition 10 (Transition Relation of an EPC)**.** For an EPC the transition relation is a triple $R = M \rightarrow N \times M$ where M refers to the marking space of the EPC. A single transition $(m, n, m') \in R$ represents a marking change of an EPC. Furthermore, we define the following notations:

- $m_1 \xrightarrow{n} m_2$: node n is enabled in marking m_1 and its firing results in m_2.
- $m_1 \rightarrow m_2$: there exists a node n such that $m_1 \xrightarrow{n} m_2$.
- $m_1 \xrightarrow{\tau} m_q$: the firing sequence $\tau = n_1 n_2 ... n_q$ produces from marking m_1 the new state m_q with $m_1 \xrightarrow{n_1} m_2, m_2 \xrightarrow{n_2} ... \xrightarrow{n_q} m_q$.
- $m_1 \xrightarrow{*} m_q$: there exists a sequence τ such that $m_1 \xrightarrow{\tau} m_q$. In this case m_q is called *reachable* from m_1.

Definition 11 (Initial Marking of an EPC)**.** For an EPC an initial marking $i \in M$ is defined as a state and context mapping that fulfills the following constraints[3]:

- $\exists a_s \in A_s : \sigma(a_s) = +1$,

[3]Note that the state is given in terms of arcs. Intuitively, one can think of start event holding positive or negative tokens. However, the corresponding arc will formally represent this token.

- $\forall a_s \in A_s$: $\sigma(a_s) \in \{-1, +1\}$,
- $\forall a_s \in A_s$: $\kappa(a_s) = wait$ if $\sigma(a_s) = +1$ and $\kappa(a_s) = dead$ if $\sigma(a_s) = -1$, and
- $\forall a \in A_{int} \cup A_e : \kappa(a) = wait$ and $\sigma(a) = 0$.

The set of all initial markings is referred to as $I \subseteq M$.

Definition 12 (Final Marking of an EPC)**.** For an EPC and a final marking $o \in M$ is defined as a marking that fulfills the following constraints:

- $\exists a_e \in A_e$: $\sigma(a_e) = +1$ and
- $\forall a \in A_{int} \cup A_s : \sigma(a) \leq 0$.

The set of all final markings is referred to as $O \subseteq M$.

3.6 Soundness of EPCs

Soundness is an important correctness criterion for business process models first introduced in [Aal97]. The original soundness property is defined for a Workflow net, a Petri net with one source and one sink, and requires that (i) for every state reachable from the source, there exists a firing sequence to the sink (option to complete); (ii) the state with a token in the sink is the only state reachable from the initial state with at least one token in it (proper completion); and (iii) there are no dead transitions [Aal97]. For EPCs, this definition cannot be used directly since EPCs may have multiple start and end events. Based on the definitions of the initial and final state of an EPC, we define soundness of an EPC analogously to soundness of Workflow nets [Aal97].

Definition 13 (Soundness of an EPC)**.** An EPC is sound if there is a set of initial markings I such that:

(i) For each start-arc a_s there exists an initial marking $i \in I$ where the arc (and hence the corresponding start event) holds a positive token. Formally:
$\forall a_s \in A_s : \exists i \in I : \sigma(a_s) = +1$

(ii) For every marking m reachable from an initial state $i \in I$, there exists a firing sequence leading from marking m to a final marking $o \in O$. Formally:
$\forall i \in I : \forall m \in M \ (i \xrightarrow{*} m) \Rightarrow \exists o \in O \ (m \xrightarrow{*} o)$

(iii) The final markings $o \in O$ are the only markings reachable from a marking $i \in I$ such that there is no node that can fire. Formally:
$\forall m \in M : \nexists m'(m \rightarrow m') \Rightarrow m \in O$

Given this definition, the EPCs of Figures 2 and Figure 3 are not sound since the OR-joins block each other. Both EPCs of Figure 4 are sound. Finally, both EPCs of Figure 5 are not sound because if the token at $a7$ or $a7f$, resp., exits the loop, the OR-join $c1$ is blocked.

4 Summary

In this paper, we revisited existing work on formalization of EPCs and OR-joins in general. We found that one of the disadvantages of these solutions is that introducing an OR-block in the process might yield unexpected behavior. Against this background, we presented a concept for the definition of EPC semantics based on state and context. In future research, we aim to provide formal semantics for the ideas presented in this paper. We also aim to support it with efficient verification techniques and apply it to a range of real-life EPCs to see if these semantics match the intuition of the modeler.

References

[Aal97] W. M. P. van der Aalst. Verification of Workflow Nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *LNCS*, pages 407–426. 1997.

[Aal99] W. M. P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639–650, 1999.

[ADK02] W. M. P. van der Aalst, J. Desel, and E. Kindler. On the semantics of EPCs: A vicious circle. In M. Nüttgens and F. J. Rump, editor, *Proc. of the 1st GI-Workshop EPK 2002*, pages 71–79, 2002.

[AH05] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[CK05] N. Cuntz and E. Kindler. On the semantics of epcs: Efficient calculation and simulation. In W. M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Business Process Management, 3rd Int. Conference BPM 2005*, volume 3649 of *LNCS*, pages 398–403, 2005.

[CS94] R. Chen and A. W. Scheer. Modellierung von Prozessketten mittels Petri-Netz-Theorie. Heft 107, Institut für Wirtschaftsinformatik, Saarbrücken, Germany, 1994.

[Cun04] Nicolas Cuntz. Über die effiziente Simulation von Ereignisgesteuerten Prozessketten. Master's thesis, Univ. of Paderborn, June 2004. (in German).

[DA04] Juliane Dehnert and Wil M. P. van der Aalst. Bridging The Gap Between Business Models And Workflow Specifications. *Int. J. Cooperative Inf. Syst.*, 13(3):289–332, 2004.

[FS01] A. Finkel and Ph. Schnoebelen. Well-structured Transition Systems everywhere! *Theoretical Computer Science*, 256(1–2):63–92, April 2001.

[HOS+06] K. van Hee, O. Oanea, A. Serebrenik, N. Sidorova, and M. Voorhoeve. Workflow model compositions perserving relaxed soundness. In S. Dustdar, J. . Fiadeiro, and A. Sheth, editors, *Business Process Management, 4th Int. Conference, BPM 2006*, volume 4102 of *LNCS*, pages 225–240. 2006.

[Kin06] Ekkart Kindler. On the semantics of EPCs: Resolving the vicious circle. *Data Knowl. Eng.*, 56(1):23–40, 2006.

[KNS92] G. Keller, M. Nüttgens, and A.-W. Scheer. Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)". Heft 89, Institut für Wirtschaftsinformatik, Saarbrücken, Germany, 1992.

[LA94] Frank Leymann and Wolfgang Altenhuber. Managing business processes an an information resource. *IBM Systems Journal*, 33(2):326–348, 1994.

[LSW98] P. Langner, C. Schneider, and J. Wehler. Petri Net Based Certification of Event driven Process Chains. In J. Desel and M. Silva, editor, *Application and Theory of Petri Nets*, volume 1420 of *LNCS*, pages 286–305, 1998.

[MMN06] Jan Mendling, Michael Moser, and Gustaf Neumann. Transformation of yEPC Business Process Models to YAWL. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing*, volume 2, pages 1262–1267, Dijon, France, 2006. ACM.

[MN03] J. Mendling and M. Nüttgens. EPC Modelling based on Implicit Arc Types. In M. Godlevsky and S. W. Liddle and H. C. Mayr, editor, *Proc. of the 2nd International Conference on Information Systems Technology and its Applications (ISTA)*, volume 30 of *LNI*, pages 131–142, 2003.

[NR02] M. Nüttgens and F. J. Rump. Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK). In J. Desel and M. Weske, editor, *Proceedings of Promise 2002, Potsdam, Germany*, volume 21 of *LNI*, pages 64–77, 2002.

[Rit00] P. Rittgen. Paving the Road to Business Process Automation. In *Proc. of the Europ. Conf. on Information Systems (ECIS)*, pages 313–319, 2000.

[Rum99] F. J. Rump. *Geschäftsprozessmanagement auf der Basis ereignisgesteuerter Prozessketten - Formalisierung, Analyse und Ausführung von EPKs*. Teubner Verlag, 1999.

[WAHE06] M.T. Wynn, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Verifying Workflows with Cancellation Regions and OR-joins: An Approach Based on Reset Nets and Reachability Analysis. In S. Dustdar, J.L. Fiadeiro, and A. Sheth, editors, *Proceedings of BPM 2006*, volume 4102 of *LNCS*, pages 389–394, Vienna, Austria, 2006.

[WEAH05] M.T. Wynn, D. Edmond, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Achieving a General, Formal and Decidable Approach to the OR-join in Workflow using Reset nets. In G. Ciardo and P. Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536, pages 423–443, 2005.