# Getting Rid of the OR-Join in Business Process Models

J. Mendling
Business Process Management group, Queensland University of Technology, Australia.
j.mendling@qut.edu.au

B.F. van Dongen
Department of Computer Science, Eindhoven University of Technology, The Netherlands.
b.f.v.dongen@tue.nl

W.M.P. van der Aalst
Department of Computer Science, Eindhoven University of Technology, The Netherlands.
w.m.p.v.d.aalst@tue.nl

## Abstract

*In practice, the development of process-aware information systems suffers from a gap between conceptual business process models and executable workflow specifications. Because of this gap, conceptual models are hardly reused as execution templates. In this paper, we address the notorious "OR-join problem" that is partly responsible for this gap. At the conceptual level people frequently use OR-joins. However, given their non-local semantics, OR-joins cannot be mapped easily onto executable languages. In particular, we present a new approach to map a conceptual process model with OR-joins (expressed in terms of an EPC) onto an executable model without OR-joins (expressed in terms of a Petri net).*

*Although we used an EPC process model as a running example, the approach is equally applicable to other process modeling languages that offer OR-joins as (e.g. BPMN). Moreover, the resulting Petri net can be mapped onto other execution languages such as BPEL. All of this has been implemented in the context of the ProM framework.*

## 1. Introduction

In recent years, business process modeling is increasingly used for the development of Process-Aware Information Systems (PAIS) [16] on an enterprise scale. In this context, business process models can serve (1) as a conceptual representation of the system such as in case of the SAP Reference Model [19], or (2) as a specification of an executable workflow process, if web service compositions are defined, for example, with BPEL [4]. While both these modeling scenarios should smoothly work together in theory, it can be observed that there is a gap in practice (see e.g. [33, p.141]) such that conceptual models are hardly reused in the implementation phase. A better integration of both application scenarios is not only desirable for streamlining the design process, but also for a more consistent business evaluation of processes that are executed by workflow systems.

There are several explanations for the gap between conceptual process modeling and executable workflow specification: the divergent terminology of business analyst and systems engineers, the different degree of technical detail, and the heterogeneous modeling tools that are used. These issues have in common that they can be soothed, at least partially, by training people, standardizing terms and concepts, or specifying interfaces. Beyond that there is one specific problem that is of a theoretical nature: several business process modeling languages, such as Event-driven Process Chains (EPC) [18] or BPMN [31], offer an OR-join synchronization element that cannot be directly transformed to most executable languages. The reason is the non-local semantics of the OR-join.

In Figure 1, activity *act* is preceded by an OR-join ($\vee$

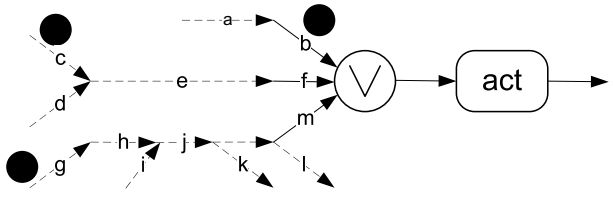


**Figure 1. Non-local semantics of the OR-join**

denotes an OR in EPC notation). This means that it can only be executed if there is at least one input and no more inputs can arrive. The black dot on the arc labeled *b* in Figure 1 is called a token and indicates that the OR-join is activated. As sketched, more tokens can arrive via the other two input arcs of the OR-join. The token on arc *c* can travel to *f* and the token on arc *g* can travel to *m*. Therefore, *act* cannot be executed yet. Suppose that the token on *c* moves to *f*. Then the OR-join still blocks, waiting for the token on arc *g*. If the token on *g* moves to *m*, *act* can be executed. Moreover, even if the token on *g* moves to *k* or *l* (i.e., the token "exits" the path to the OR-join), then *act* can be executed, in which case, the OR-join only consumes two tokens because it can be sure that a third token will not arrive via input arc *m*.

Because of the tricky non-local semantics, the OR-join is not supported by most execution languages. Many systems use a Petri-net-like execution engine not allowing for OR-joins. Even BPEL engines that support dead path elimination (cf. [24]) can only handle OR-join behavior in acyclic flow activities. Engines that provide full support for the OR-join (e.g. the YAWL engine [2]) may run into performance problems when there are many non-trivial OR-joins. Furthermore, in [28, 29], it was shown that the use of OR-joins in EPCs constitute to most of the errors in models, i.e. just by counting the number of OR-joins in an EPC, it is possible to predict with great accuracy whether that EPC contains errors.

Due to these reasons, process models with OR-joins often require substantial rework before they can be used in workflow systems. Since OR-joins tend to be included in every third EPC model [25, p.205], this is a serious limitation. Therefore, it is desirable to automate the removal of OR-joins for execution.

In this paper, we present a rigorous and automatic approach for getting rid of the OR-join in conceptual process models without changing the semantics. We use EPCs a conceptual modeling language and Petri nets as an execution language to illustrate the procedure. However, it must be noted that the approach is similarly applicable if, for example, BPMN is used as a conceptual language and BPEL as an implementation language. In particular, we combine a recent EPC semantics definition with tools for Petri net synthesis. Our contribution in this context is the innovative combination of these complementary theories and the provision of respective tool support within the ProM framework.

Against this background, the paper is structured as follows. In Section 2, we introduce Event-driven Process Chains (EPCs). Furthermore, we illustrate the calculation of the reachability graph for an EPC and present its implementation within the ProM framework. In Section 3, we give an overview of Petri net synthesis and demonstrate how the tool Petrify by *Cortadella et al.* [11] is used to build a Petri net from an EPC with OR-joins. Petrify has been embedded into the ProM framework and the results presented in this paper are therefore fully implemented in ProM. Moreover, ProM allows for the *automatic translation of the resulting Petri net to other notations such as EPCs, YAWL, BPEL, etc.*, thus enabling the application of our results in a larger context. Section 4 discusses our approach in the light of related work, in particular on transformation of process models with OR-joins. Finally, Section 5 concludes the paper.

## 2. Event-driven Process Chains (EPCs)

This section uses an example to introduce the EPC notation and syntax. Then, Section 2.3 discusses the formalization of EPC semantics, followed by Section 2.4 which presents a ProM plug-in for calculating the reachability graph.

### 2.1. Introductory EPC Example

The Event-driven Process Chain (EPC) is a business process modeling language for the representation of temporal and logical dependencies of activities in a business process (see [18]). EPCs offer *function type* elements to capture activities of a process and *event type* elements describing pre-conditions and post-conditions of functions. Furthermore, there are three kinds of *connector types* including AND (symbol $\wedge$), OR (symbol $\vee$), and XOR (symbol $\times$) 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. Furthermore, OR- and XOR-splits after events are not allowed, since events cannot make decisions. Control flow arcs are used to link these elements.

The informal (or intended) semantics of an EPC can be described as follows. The AND-split activates all subsequent branches in concurrency. 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, splits from an event to multiple functions are forbidden with XOR and OR as the activation conditions do not become clear in the model. The AND-join waits for all incoming 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. This feature is called non-locality, since the state of all transitive predecessor nodes has to be considered.

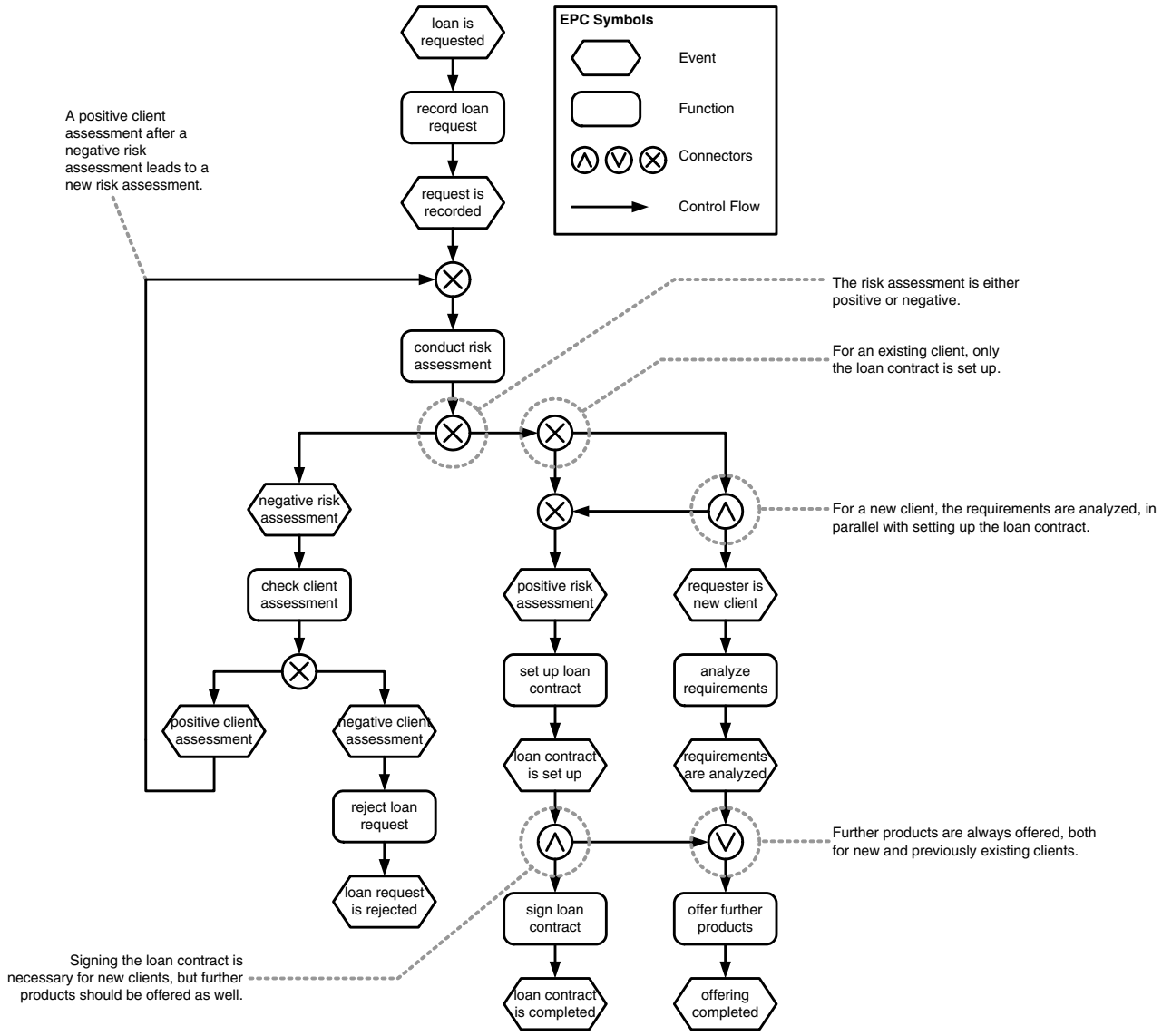


**Figure 2. EPC for a loan request process [30]**

Figure 2 shows an EPC model for a loan request process as described in *Nüttgens and Rump* [30]. The start event *loan is requested* signals the start of the process and the precondition to execute the *record loan request* function. After the postcondition *request is recorded*, the process continues with the function *conduct risk assessment* after the XOR-join connector. The subsequent XOR-split connector indicates a decision. In case of a *negative risk assessment*, the function *check client assessment* is performed. The following second XOR-split marks another decision: in case of a *negative client assessment*, the process ends with a rejection of the loan request; in case of a *positive client assessment* the *conduct risk assessment* function is executed a second time under consideration of the positive client assessment. If the risk assessment is not negative, there is another decision point to distinguish new clients and existing clients. In case of an existing client, the *set up loan contract* function is conducted. After that, the AND-split indicates that two activities have to be executed: first, the *sign loan contract* function, and second the *offer further products* function. If the client is new, the *analyze requirements* function has to be performed in addition to setting up the loan contract. The OR-join waits for both functions to be completed if necessary. If the *analyze requirements* function will not be executed in the process, it continues with the function *offer further products* immediately, otherwise it synchronizes the two branches.

## 2.2. EPC Syntax

In this section, we define the syntax of EPCs in line with the previous introduction and the presentation in [25].

**Definition 2.1. (EPC)** A flat $EPC = (E, F, C, l, A)$ consists of four pairwise disjoint and finite sets $E, F, C$, a mapping $l : C \rightarrow \{and, or, xor\}$, and a binary relation $A \subseteq (E \cup F \cup C) \times (E \cup F \cup C)$ 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 $C$ 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 $N = E \cup F \cup C$ is called a *node*.

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

**Definition 2.2. (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 $\bullet n = \{x \in N | (x, n) \in A\}$ as its *preset*, and $n\bullet = \{x \in N | (n, x) \in A\}$ as its *postset*.

**Definition 2.3. (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 2.4. (Paths and Connector Chains)** Let $EPC = (E, F, C, l, A)$ be a flat EPC and $a, b \in N$ be two of its nodes. A *path* $a \hookrightarrow b$ refers to the existence of a sequence of EPC 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 $a \neq b \in N$ 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 $(a, b) \in A$.

**Definition 2.5. (Subsets)** For any $EPC = (E, F, C, l, A)$, we define the following (implicit) subsets of nodes and arcs:

- $E_s = \{e \in E \mid |\bullet e| = 0 \wedge |e\bullet| = 1\}$ is the set of start-events,
  $E_{int} = \{e \in E \mid |\bullet e| = 1 \wedge |e\bullet| = 1\}$ is the set of intermediate-events, and
  $E_e = \{e \in E \mid |\bullet e| = 1| \wedge e\bullet| = 0\}$ is the set of end-events.
- $J = \{c \in C \mid |\bullet c| > 1$ and $|c\bullet| = 1\}$ is the set of join-connectors, and
  $S = \{c \in C \mid |\bullet c| = 1$ and $|c\bullet| > 1\}$ is the set of split-connectors.
- $C_{and} = \{c \in C \mid l(c) = and\}$ is the set of and-connectors,
  $C_{xor} = \{c \in C \mid l(c) = xor\}$ is the set of xor-connectors, and
  $C_{or} = \{c \in C \mid l(c) = or\}$ is the set of or-connectors.
- $A_s = \{(x, y) \in A \mid x \in E_s\}$ is the set of start-arcs,
  $A_{int} = \{(x, y) \in A \mid x \notin E_s \wedge y \notin E_e\}$ is the set of intermediate-arcs, and
  $A_e = \{(x, y) \in A \mid y \in E_e\}$ is the set of end-arcs.

Several notions of syntactical correctness have been proposed, see [25]. We use a rather small set of requirements in order to provide semantics for a large set of EPCs.

**Definition 2.6. (Syntactically Correct EPC)** An $EPC = (E, F, C, l, A)$ is called syntactically correct, if it fulfills the requirements:

1. $EPC$ is a directed and coherent graph such that $\forall_{n \in N}\ \exists_{e_1 \in E_s, e_2 \in E_e}\ e_1 \hookrightarrow n \hookrightarrow e_2$.
2. There is at least one start node and one end node in an EPC: $|E_s| \geq 1 \wedge |E_e| \geq 1$.
3. Events have at most one incoming and one outgoing arc: $\forall_{e \in E}\ |\bullet e| \leq 1 \wedge |e\bullet| \leq 1$.
4. Functions have exactly one incoming and one outgoing arcs: $\forall_{f \in F}\ |\bullet f| = 1 \wedge |f\bullet| = 1$.
5. Connectors are splits or joins, i.e., $\forall_{c \in C}\ (|\bullet c| = 1 \wedge |c\bullet| \geq 1) \vee (|\bullet c| \geq 1 \wedge |c\bullet| = 1)$.

In the remainder we assume EPCs to be syntactically correct.

## 2.3. EPC Semantics

Several semantical definitions have been proposed for EPCs, see *Kindler* [21] for an overview and a theoretical framework. In this paper, we use a recent formalization of EPC semantics as proposed by *Mendling and van der Aalst* [26], that has some desirable properties compared to earlier approaches. Still, our general approach is equally applicable if other formalizations such as the ones presented in [2, 21] are considered.

The principal idea of the semantics by *Mendling and van der Aalst* lends some concepts from *Langner, Schneider, and Wehler* [22], and adapts the idea of Boolean nets with true and false tokens in an appropriate manner. The reachability graph, that we will formalize later, depends on the state and the context of an EPC. The *state* of an EPC is basically an assignment 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. 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 utilized by an OR-join to determine whether it has to synchronize with further positive tokens or not. In contrast to Petri nets, we distinguish the terms *marking* and *state*.

**Definition 2.7. (State and Context)** For an $EPC = (E, F, C, l, A)$, the mapping $\sigma : A \to \{-1, 0, +1\}$ is called a *state* 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 $\kappa : A \to \{wait, dead\}$ 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.

**Definition 2.8. (Marking of an EPC)** For a syntactically correct $EPC$ the mapping $m : A \to \{-1, 0, +1\} \times \{wait, dead\}$ is called a *marking*. The set of all markings $M_{EPC}$ of an EPC is called *marking space* with $M_{EPC} = A \times \{-1, 0, +1\} \times \{wait, dead\}$. The projection of a given marking $m$ to a subset of arcs $S \subseteq A$ is referred to as $m_S$. If we refer to the $\kappa$- or the $\sigma$-part of $m$, we write $\kappa_m$ and $\sigma_m$, respectively, i.e., $m(a) = (\sigma_m(a), \kappa_m(a))$.

The propagation of context status and state tokens is arranged in a four phase cycle: (1) dead context, (2) wait context, (3) negative token, and (4) positive token propagation. Whether a node is enabled and how it fires is illustrated in Figure 3. A formalization of the transitions for each phase is presented in [25].

1. In the first phase, all *dead context* information is propagated in the EPC until no new dead context can be derived.
2. 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 (see [25]).
3. After that, all *negative tokens* are propagated until no negative token can be propagated anymore. This phase cannot run into an endless loop (see [25]).
4. Finally, one of the enabled nodes is selected and propagated *positive tokens* leading to a new iteration of the four phase cycle.

The transition relations of the four phases have been formally defined in [25]. They provide the intermediate results for calculating the reachability graph.

## 2.4. Reachability Graph Calculation

Initial and final markings are the start and end points for calculating the reachability graph of an EPC. We define the sets of initial and the final markings similar to the definition by *Rump* [32]. An initial marking is an assignment of positive or negative tokens to all start arcs while all other arcs have no token, and in a final marking only end arcs may hold positive tokens.

**Definition 2.9. (Initial Marking of an EPC)** Let $EPC = (E, F, C, l, A)$ be a syntactically correct EPC and $M_{EPC}$ its marking space. $I_{EPC} \subseteq M_{EPC}$ is defined as the set of all possible initial markings, i.e., $m \in I_{EPC}$ if and only if:

- $\exists_{a_s \in A_s}\ \sigma_m(a_s) = +1$,
- $\forall_{a_s \in A_s}\ \sigma_m(a_s) \in \{-1, +1\}$,
- $\forall_{a_s \in A_s}\ \kappa_m(a_s) = wait$ if $\sigma_m(a_s) = +1$ and $\kappa_m(a_s) = dead$ if $\sigma_m(a_s) = -1$, and
- $\forall_{a \in A_{int} \cup A_e}\ \kappa_m(a) = wait$ and $\sigma_m(a) = 0$.

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

**Definition 2.10. (Final Marking of an EPC)** Let $EPC = (E, F, C, l, A)$ be a syntactically correct EPC and $M_{EPC}$ its marking space. $O_{EPC} \subseteq M_{EPC}$ is defined as the set of all possible final markings, i.e., $m \in O_{EPC}$ if and only if:

- $\exists_{a_e \in A_e}\ \sigma_m(a_e) = +1$, and
- $\forall_{a \in A_s \cup A_{int}}\ \sigma_m(a) \leq 0$.

In this context, a marking $m'$ is called reachable[1] from another marking $m$ if and only if, after applying the phases of dead and wait context and negative token propagation on $m$, there exists a node $n$ whose firing in the positive token propagation phase produces $m'$. This is denoted by $m \xrightarrow{n} m'$. The notation $m \to m'$ indicates that there is some $n$ such that $m \xrightarrow{n} m'$. Furthermore, we write $m_1 \xrightarrow{\tau} m_q$ if there is a firing sequence $\tau = n_1 n_2 ... n_{q-1}$ that produces from marking $m_1$ the new marking $m_q$ with $m_1 \xrightarrow{n_1} m_2, m_2 \xrightarrow{n_2} ... \xrightarrow{n_{q-1}} m_q$. If there exists a sequence $\tau$ such that $m_1 \xrightarrow{\tau} m_q$, we write $m_1 \xrightarrow{*} m_q$. Accordingly, we define the reachability graph $RG$ as follows.

[1] A formalization of *reachability* is given in [25].

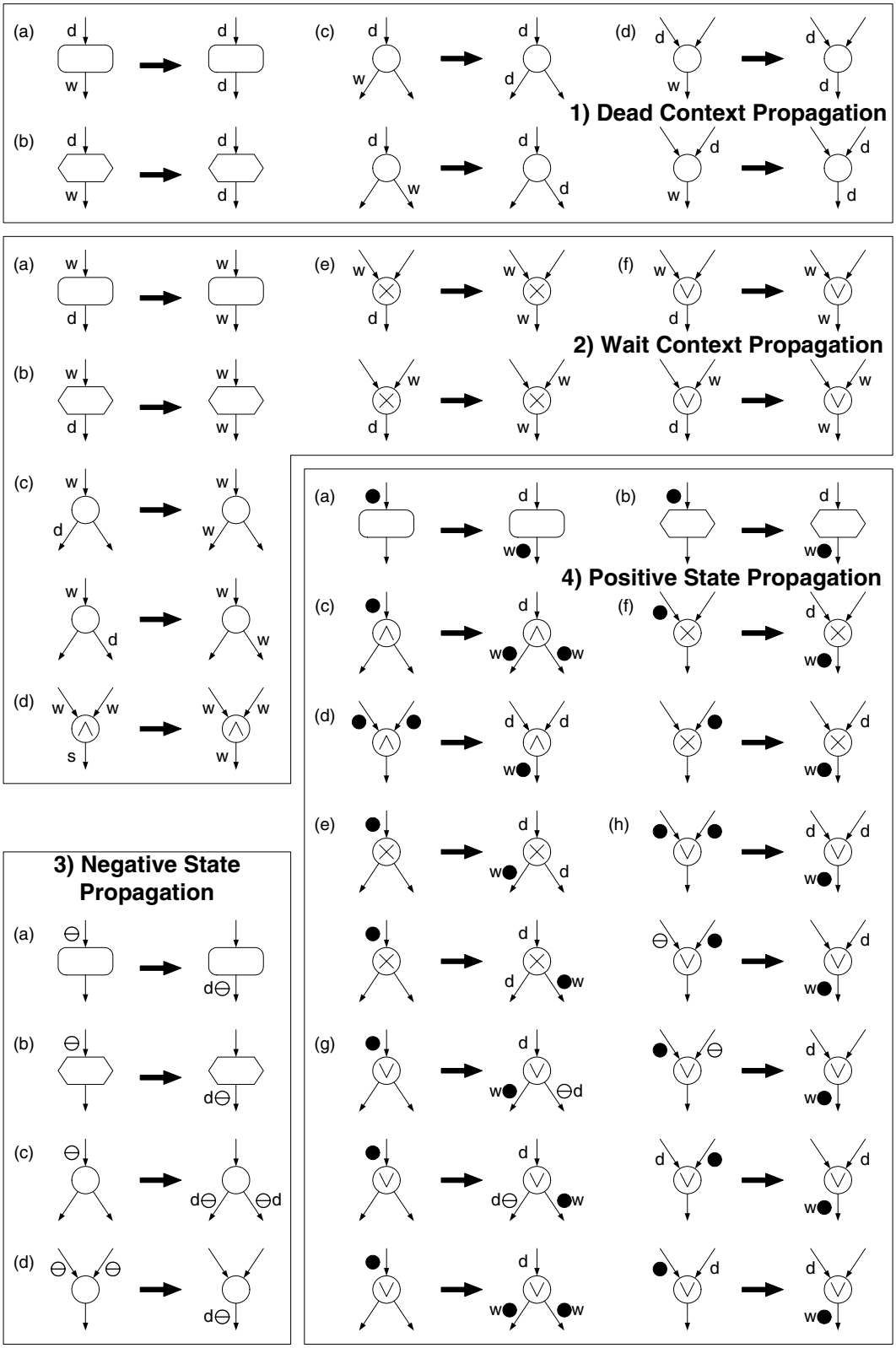


**Figure 3. Propagation of context and state in four phases**

**Algorithm 1** Pseudo code for calculating the reachability graph of an EPC

**Require:** $EPC = (E, F, C, l, A), I \subseteq I_{EPC}$

1: $RG \leftarrow \emptyset$
2: $toBePropagated \leftarrow I$
3: $propagated \leftarrow \emptyset$
4: **while** $toBePropagated \neq \emptyset$ **do**
5: $currentMarking \leftarrow toBePropagated.pop()$
6: $oldMarking \leftarrow currentMarking.clone()$
7: $currentMarking.propagateDeadContext(EPC)$
8: $currentMarking.propagateWaitContext(EPC)$
9: $currentMarking.propagateNegativeTokens(EPC)$
10: $nodeNewMarking \leftarrow currentMarking.propagatePositiveTokens(EPC)$
11: $propagated.add(oldMarking)$
12: **for all** $(node, newMarking) \in nodeNewMarkings$ **do**
13: $RG.add(oldMarking, node, newMarking)$
14: **if** $newMarking \notin propagated$ **then**
15: $toBePropagated.push(newMarking)$
16: **end if**
17: **end for**
18: **end while**
19: **return** $RG$

**Definition 2.11. (Reachability Graph of an EPC)** Let $EPC = (E, F, C, l, A)$ be a syntactically correct EPC, $N = E \cup F \cup C$ its set of nodes, and $M_{EPC}$ its marking space. Then, the reachability graph $RG \subseteq M_{EPC} \times N \times M_{EPC}$ of an EPC contains the following nodes and transitions:

(i) $\forall_{m \in I_{EPC}} m \in RG$.

(ii) $(m, n, m') \in RG$ if and only if $m \xrightarrow{n} m'$.

The calculation of $RG$ requires an $EPC$ and a set of initial markings $I \subseteq I_{EPC}$ as input. For several EPCs from practice, such a set of initial markings will not be available. In this case, one can easily calculate the set of all possible initial markings, as shown in [29]. Algorithm 1 uses an object-oriented pseudo code notation to define the calculation. In particular, we assume that $RG$ is an instance of the class $ReachabilityGraph$, $propagated$ an instances of class $Set$, and $toBePropagated$ an instance of class $Stack$ that provides the methods $pop()$ and $push()$. Furthermore, $currentMarking$, $oldMarking$, and $newMarking$ are instances of class $Marking$ that provides the methods $clone()$ to return a new, but equivalent marking, $propagateDeadContext(EPC)$, $propagateWaitContext(EPC)$, as well as method $propagateNegativeTokens(EPC)$ to change the marking according to the transitions of the respective phase, i.e., to determine $max_d, max_w$, and $max_{-1}$ of the current marking. Finally, $propagatePositiveTokens(EPC)$ returns a set of $(node, marking)$ pairs including the node that can fire and the marking that is reached after the firing.

In lines 1-3, the sets $RG$ and $propagated$ are initialized with the empty set, and the stack $toBePropagated$ is filled with all initial markings of the set $I$. The while loop between lines 4-18 calculates new markings for the marking that is on top of the stack $toBePropagated$. In particular, $currentMarking$ receives the top marking from the stack (line 5) and it is cloned into the $oldMarking$ object (line 6). In lines 7-9, the propagations of dead and wait context and of negative tokens are applied on $currentMarking$. Then, in line 10, the pairs of nodes and new markings that can be reached from the old marking are stored in the set $nodeNewMarking$. After that, the old marking is added to the $propagated$ set (line 11). In lines 12-17, for each pair of node and new marking, a new transition $(oldMarking, node, newMarking)$ is added to $RG$. If a new marking was not yet propagated, it is pushed on top of the $toBePropagated$ stack (lines 14-16). Using a stack, the reachability graph is calculated in a depth-first manner (where the termination is guaranteed by filling the set $propagated$). Finally, in line 19, $RG$ is returned.

Based on the previous algorithm, we have implemented a conversion plug-in for the ProM (Process Mining) framework [7, 15]. This conversion plug-in calculates the reachability graph of an EPC and displays it as a transition systems. Figure 4 shows the reachability graph of the example EPC that is generated by using the plug-in (the three highlighted parts can be ignored for now, they will be addressed in the next section). The densely connected part looking like a diamond in the center of the reachability graph stems from two concurrent paths of the EPC after the AND-split. These

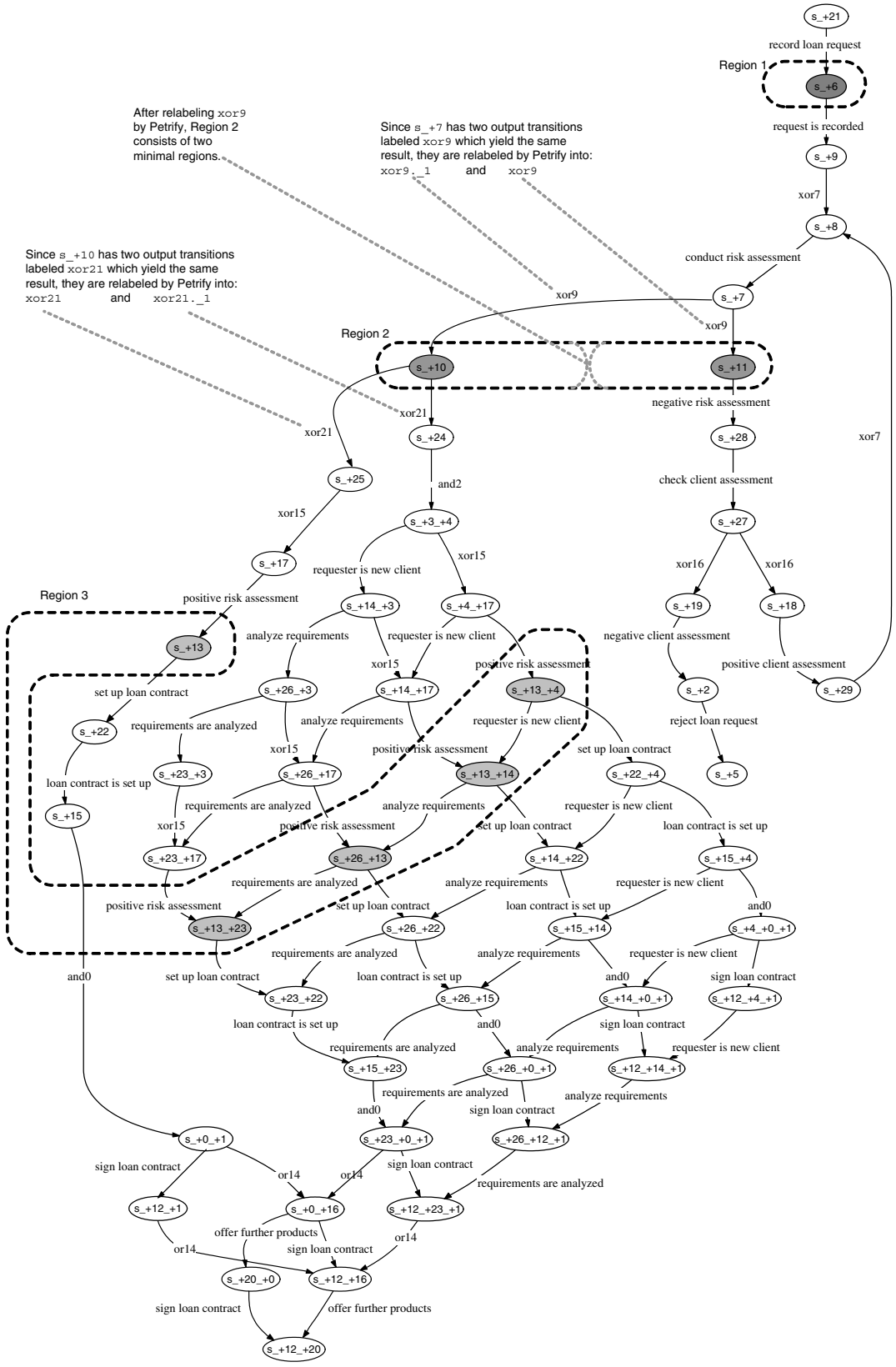


**Figure 4. Reachability graph of the EPC for the loan request process, showing three of its regions**

paths are eventually synchronized by an OR-join. This OR-join can be found as the label or14 on two arcs exiting from the diamond at the bottom. Since ProM can import a variety of EPC interchange formats, the reachability graph calculation can be easily used for existing models.

## 3. Petri Net Synthesis

In this section, we present the second step of our approach. In this step, a Petri net is synthesized from the reachability graph resulting from the first step. To do this we use the "Theory of Regions" [12, 14, 17]. However, we first introduce Petri nets in more detail.

### 3.1. Petri nets

A Petri net consists of two modeling elements, namely *places* and *transitions*. These elements are the nodes of a bipartite graph, partitioned into places and transitions. When a Petri net is represented visually, we draw transitions as boxes and places as circles. A transition typically corresponds to either an activity which needs to be executed, or to a "silent" step that takes care of routing. Places are typically used to define the preconditions and postconditions of transitions. In this paper, we do not show the labels of places, since they do not correspond to active elements of a Petri net.

Transitions and places are connected through directed arcs in such a way that the places and transitions make up the partitions in a bipartite graph (no place is connected to a place and no transition is connected to a transition). The places that are in the pre-set of a transition are called its input places and the places in the post-set of a transition its output places.

To denote the state of a process execution the concept of *tokens* is used. A token is placed inside a place to show that a certain condition holds. Each place can contain arbitrarily many of such tokens. When a transition execution occurs (in other words, a transition *fires*), one token is removed from each of the input places and one token is produced for each of the output places. Note that this restricts the behavior in such a way that a transition can only occur when there is at least one token in *each* of the input places. The distribution of tokens over the places is called a *state*, better known as *marking* in Petri net jargon.

### 3.2. Theory of Regions

The Theory of Regions was developed to transform *state-based models*, such as transition systems or reachability graphs, to *Petri nets*, in such a way that the behavior of the resulting Petri net *exactly* corresponds to the original state-based model. For this purpose, the concept of *regions* was introduced in [17], where these regions served as intermediate objects, between a transition system on the one hand and a Petri net on the other hand. The process of going from a state-based model to a Petri net, is called *synthesis*.

Before we formalize the concept of a region, we first provide an insight into their meaning using our example of Figure 4. A region represents a collection of states, such that (1) if a certain transition $a$ enters the region, then all transitions $a$ enter the region and (2) if a certain transition $b$ exits a region, then all transitions $b$ exit the region. Figure 4 shows this concept using the reachability graph of Figure 4. It shows three regions, namely:

1. Region 1 contains only one state such that all transitions *record loan request* enter the region and all transitions *request is recorded* exit the region,
2. Region 2 contains two states such that all transitions *xor9* enter the region and all transitions *xor21* and *negative risk assessment* exit the region,
3. Region 3 contains five states, such that all transitions *positive risk assessment* enter the region and all transitions *set up loan contract* exit the region. Furthermore, the transitions *requester is new client*, *analyze requirements* and *requirements are analyzed* appear both inside, as well as outside the region, but they do not *cross* the region (i.e. enter or exit the region).

Regions, such as the three examples but also all others, have a special meaning in the state-based model, i.e., they show causal dependencies between the transitions. Consider for example Region 3. From that region, it becomes clear that after you perform a *positive risk assessment*, you at some point perform *set up loan contract*. However, in the mean time, many other transitions may occur. Using this type of reasoning, a collection of regions can be translated into a Petri net which models the exact behavior of the state-based model.

**Definition 3.1. (Region)** Let $TS = (S, \Lambda, \rightarrow)$ be a state-based model ($S$ is the set of states, $\Lambda$ the set of transitions and $\rightarrow$ the transition relation, i.e., the labeled transitions between states).[2] We say that $R \subseteq S$ is a region of $TS$ if and only if for all $(p, \alpha, q), (p', \alpha, q') \in \rightarrow$ holds that:

- if $p \in R$ and $q \notin R$ then $p' \in R$ and $q' \notin R$, i.e., all transitions labeled $\alpha$ *exit* the region, and we say that $R$ is a *pre-region* of $\alpha$,
- if $p \notin R$ and $q \in R$ then $p' \notin R$ and $q' \in R$, i.e., all transitions labeled $\alpha$ *enter* the region, and we say that $R$ is a *post-region* of $\alpha$.
- if $(p \in R) = (q \in R)$ then $(p' \in R) = (q' \in R)$, i.e., all transitions labeled $\alpha$ *do not cross* the region.

[2] Note that any reachability graph defines a state-based model $TS$.

It is easy to see that there are two *trivial* regions, i.e., $\emptyset \subseteq S$ and $S \subseteq S$ are regions. The collection of all regions of a transition system $TS$ is called $\Re(TS)$. A region $R \in \Re(TS)$ is said to be *minimal* if and only if for all $R' \subset R$ with $R' \neq \emptyset$ holds that $R' \notin \Re(TS)$, i.e., none of its subsets is also a region. The set of all minimal regions is denoted by $\Re^{min}(TS)$. Furthermore, it is important to note that regions do not depend on one label $\alpha$, i.e., they always depend on the entire set of labels in the transition system.

### 3.3. Synthesis of EPC Reachability Graph

As we explained before, the intuition behind a region is that it represents a causal dependency between the transitions that enter the region and the transitions that exit the region, and hence for Petri net synthesis, a region corresponds to a *Petri net place* and a transition corresponds to a *Petri net transition*. Thus, the main idea of the *synthesis algorithm* is the following: for each transition $t$ in the state-based model a transition labeled with $t$ is generated in the Petri net. For each minimal region $r_i$ a place $p_i$ is generated[3]. The flow-relation of the Petri net is constructed as follows. For each transition $t$ that enters region $r_i$, there is an edge from $t$ to $p_i$ in the Petri net and for every transition $t$ that exits $r_i$, there is an edge from $p_i$ to $t$ in the Petri net.

The Petri net synthesized from state space generated by the EPC is given in Figure 5, where some places are highlighted. The incoming place of the transition *request is recorded* corresponds to Region 1 in Figure 4. Region 3 of Figure 4 corresponds to the place between *positive risk assessment* and *set up loan contract* in the Petri net.

The first papers on the Theory of Regions only dealt with a special class of state-based models called *elementary transition systems* [5, 6, 14]. Region 2 of Figure 4 is an example of a situation that is not allowed in such an elementary transition system. The state `S_+10` has two outgoing transitions, both labeled `xor21`. Since they both result in different states, they cannot represent the same transition, and the algorithms implemented in Petrify therefore split these two transitions into `xor21` and `xor21._1`. Similarly, `xor9` is split into `xor9` and `xor9._1`. Therefore, region 2 of Figure 4 corresponds to two places in Figure 5, namely the output place of `xor9` and the output place of `xor9._1`.

In general, the class of elementary transition systems is very restricted and the state spaces we derive from our EPCs only by coincidence fall into the class of elementary transition systems. Fortunately, in the papers of Cortadella et al. [12], a method for handling *any* state-based model was presented. This approach uses *labeled Petri nets*, i.e., different transitions can refer to the same event. For this approach

[3]By using only minimal regions, the number of places is minimized. However, Petrify has many more options to generate Petri nets belonging to specific classes, such as pure or free-choice nets.

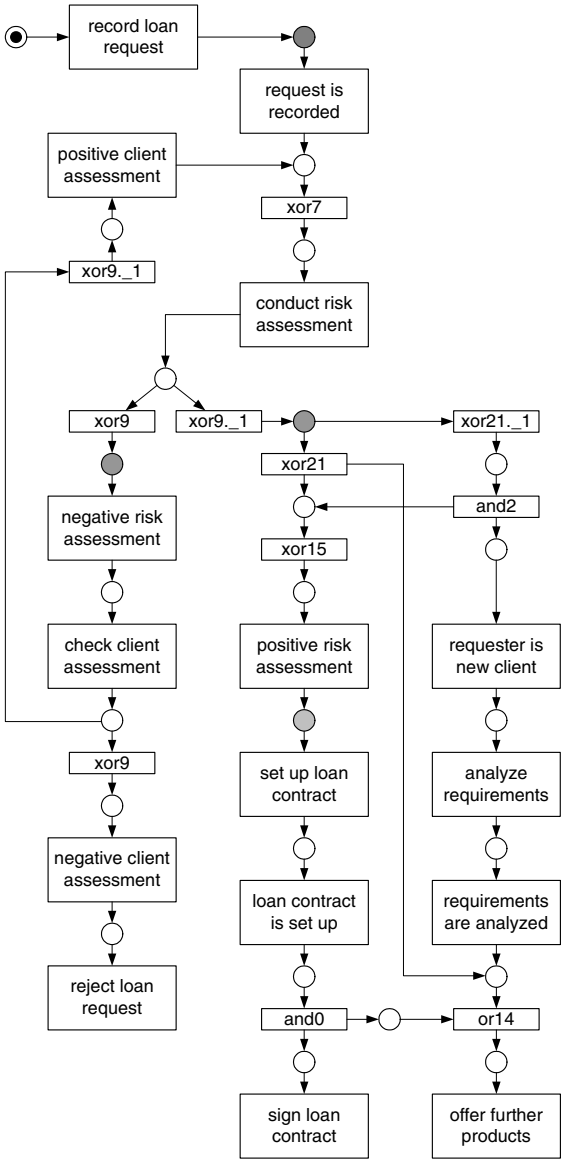


**Figure 5. Petri net for the loan request process**

it has been shown that the *reachability graph* of the synthesized Petri net is *bisimilar* to the initial state-based model.

To conclude, we mention that ProM is capable of translating this Petri net back to an EPC, in which case no OR-join connectors will be introduced, since each transition has clear AND or XOR semantics.

## 4. Related Work

Several researchers and practitioners tried to bridge the gap between conceptual process modeling and workflow execution by defining model transformations. In partic-

ular, mappings from EPCs to Petri nets were defined in [1, 9, 13, 22]. The problem of these mappings is that they are only applicable for different subclasses of EPCs, and that they do not preserve the behavior in the general case. Our approach builds on an EPC semantics definition that is applicable for all EPCs that are syntactically correct. This contrasts earlier approaches that define either semantics for the subclass of clean EPCs [21] or transformations to structured models [27, 34] which can only be applied for a subclass of some models with concurrency [20]. Furthermore, we use the Theory of Regions to derive a Petri net that is bisimilar to the original model. This Petri net can be deployed with several existing workflow engines.

We refer to [14] and [17] for the synthesis of safe Petri nets and [6] for more general cases. In these papers, the initial input describing the behavior of the process is given in the form of transition systems (where the events are known but the states are anonymous). Typically, in process mining, the observed behavior is not complete (as it is in a transition system) and it is not known, which process executions lead to which states (black box).

The application of the Theory of Regions in the context of process mining has been addressed in [3], where the authors address process mining in a software engineering setting. One of the challenges faced there is to find state information in event logs. In [3], the authors propose several ways of doing so. Furthermore, their approach is implemented in ProM by making a link between the event logs of ProM and a well-known tool tailored towards the application of the Theory of Regions, called Petrify [11].

Finally, it is worth mentioning that regions have been used in many different settings, e.g., for the synthesis and verification of asynchronous circuits [10] or for the verification of security properties [8].

## 5. Conclusions

In this paper, we presented an approach to close the gap between conceptual process models with OR-joins used when designing information systems and Petri-net-based workflow models used for the execution of such systems. This approach builds on an innovative combination of a recent formalization of process models with OR-joins and the Theory of Regions for the synthesis of Petri nets. In contrast to related work on transformations between process models, our approach yields a Petri net that exactly mimics the behavior of the original model. Although we used an EPC model as a running example, the approach is equally applicable for other modeling languages offering OR-joins, such as BPMN.

Our approach is fully implemented in the ProM framework, i.e., we have developed (1) a plug-in to construct reachability graphs of EPCs with OR-joins as described in this paper, (2) a shell to convert reachability graphs into Petri nets using Petrify, and (3) various conversions to transform Petri nets into EPCs, BPEL specifications, and YAWL models and vice versa.

In future research, we aim to provide a more direct mapping from EPCs to Petri nets based on insights from our approach. In particular, we want to investigate in how far reduction rules as defined in [23] can be used for the transformation task.

## References


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

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

[3] W.M.P. van der Aalst, V. Rubin, B.F. van Dongen, E. Kindler, and C.W. Günther. Process Mining: A Two-Step Approach using Transition Systems and Regions. BPM Center Report BPM-06-30, BPMcenter.org, 2006.

[4] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guizar, N. Kartha, C.K. Liu, R. Khalaf, D. Koenig, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web services business process execution language version version 2.0. Committee specification 31 january 2007, OASIS, 2007.

[5] E. Badouel, L. Bernardinello, and P. Darondeau. The Synthesis Problem for Elementary Net Systems is NP-complete. *Theoretical Computer Science*, 186(1-2):107–134, 1997.

[6] E. Badouel and P. Darondeau. Theory of regions. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 529–586. Springer-Verlag, Berlin, 1998.

[7] P. Barborka, L. Helm, G. Köldorfer, J. Mendling, G. Neumann, B.F. van Dongen, H.M.W. Verbeek, and W.M.P. van der Aalst. Integration of EPC-related Tools with ProM. In M. Nüttgens and F.J. Rump and J. Mendling, editor, *Proceedings of the 5th GI Workshop on Business Process Management with Event-Driven Process Chains (EPK 2006)*, pages 105–120, Vienna, Austria, December 2006. German Informatics Society.

[8] N. Busi and R. Gorrieri. A Survey on Non-interference with Petri Nets. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 328–344. Springer, 2004.

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

[10] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. A Region-based Theory for State Assignment in Speed- independent Circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 16(8):793–812, 1997.

[11] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[12] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri Nets from Finite Transition Systems. *IEEE Transactions on Computers*, 47(8):859–882, August 1998.

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

[14] J. Desel and W. Reisig. The Synthesis Problem of Petri Nets. *Acta Informatica*, 33(4):297–315, 1996.

[15] B.F. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM framework: A New Era in Process Mining Tool Support. In G. Ciardo and P. Darondeau, editors, *Application and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer-Verlag, Berlin, 2005.

[16] M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley & Sons, 2005.

[17] A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures - Part 1 and Part 2. *Acta Informatica*, 27(4):315–368, 1989.

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

[19] G. Keller and T. Teufel. *SAP(R) R/3 Process Oriented Implementation: Iterative Process Prototyping*. Addison-Wesley, 1998.

[20] B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On structured workflow modelling. In B. Wangler and L. Bergman, editors, *Advanced Information Systems Engineering, 12th International Conference CAiSE 2000, Stockholm, Sweden, June 5-9, 2000, Proceedings*, volume 1789 of *Lecture Notes in Computer Science*, pages 431–445. Springer, 2000.

[21] E. Kindler. On the semantics of EPCs: Resolving the vicious circle. *Data & Knowledge Engineering*, 56(1):23–40, 2006.

[22] 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 *Lecture Notes in Computer Science*, pages 286–305, 1998.

[23] K.B. Lassen and W.M.P. van der Aalst. WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes to Readable BPEL. In R. Meersman and Z. Tari et al., editors, *On the Move to Meaningful Internet Systems 2006, OTM Confederated International Conferences, 14th International Conference on Cooperative Information Systems (CoopIS 2006)*, volume 4275 of *Lecture Notes in Computer Science*, pages 127–144. Springer-Verlag, Berlin, 2006.

[24] F. Leymann and W. Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, 33(2):326–348, 1994.

[25] J. Mendling. *Detection and Prediction of Errors in EPC Business Process Models*. PhD thesis, Vienna University of Economics and Business Administration, 2007.

[26] J. Mendling and W.M.P. van der Aalst. Formalization and Verification of EPCs with OR-Joins Based on State and Context. In J. Krogstie, A.L. Opdahl, and G. Sindre, editors, *Proceedings of the 19th Conference on Advanced Information Systems Engineering (CAiSE 2007)*, volume 4495 of *Lecture Notes in Computer Science*, pages 439–453, Trondheim, Norway, 2007. Springer-Verlag.

[27] J. Mendling, K.B. Lassen, and U. Zdun. Transformation strategies between block-oriented and graph-oriented process modelling languages. In F. Lehner, H. Nösekabel, and P. Kleinschmidt, editors, *Multikonferenz Wirtschaftsinformatik 2006, XML4BPM Track, Band 2*, pages 297–312, Passau, Germany, February 2006. GITO-Verlag Berlin.

[28] J. Mendling, M. Moser, G. Neumann, H.M.W. Verbeek, B.F. van Dongen, and W.M.P. van der Aalst. A Quantitative Analysis of Faulty EPCs in the SAP Reference Model. BPM Center Report BPM-06-08, Eindhoven University of Technology, Eindhoven, 2006.

[29] J. Mendling, H.M.W. Verbeek, B.F. van Dongen, W.M.P. van der Aalst, and G. Neumann. Detection and Prediction of Errors in EPCs of the SAP Reference Model. *Data and Knowledge Engineering (accepted)*, 2007.

[30] 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 *Lecture Notes in Informatics*, pages 64–77, 2002.

[31] OMG, ed. Business Process Modeling Notation (BPMN) Specification. Final Adopted Specification, dtc/06-02-01, Object Management Group, February 2006.

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

[33] A.-W. Scheer, O. Thomas, and O. Adam. *Process Aware Information Systems: Bridging People and Software Through Process Technology*, chapter Process Modeling Using Event-Driven Process Chains, pages 119–146. Wiley Publishing, 2005.

[34] W. Zhao, R. Hauser, K. Bhattacharya, B.R. Bryant, and F. Cao. Compiling business processes: untangling unstructured loops in irreducible flow graphs. *Int. Journal of Web and Grid Services*, 2(1):68–91, 2006.