

Pushing the Limits: Concurrency Detection in Acyclic Sound Free-Choice Workflow Nets in $O(P^2 + T^2)$

Thomas M. Prinz^{1,*}, Julien Klaus² and Nick R.T.P. van Beest³

¹Course Evaluation Service, Friedrich Schiller University Jena, 07743 Jena Germany

²Faculty of Mathematics and Computer Science, Friedrich Schiller University Jena, 07743 Jena, Germany

³Data61, Commonwealth Scientific and Industrial Research Organisation (CSIRO), Brisbane, Australia

Abstract

Concurrency is an important aspect of Petri nets to describe and simulate the behavior of complex systems. Knowing which places and transitions could be executed in parallel helps to understand nets and enables analysis techniques and the computation of other properties, such as causality, exclusivity, etc.. All techniques based on concurrency detection depend on the efficiency of this detection methodology. Kovalyov and Esparza have developed algorithms that compute all concurrent places in $O((P+T)TP^2)$ for live and bounded nets (where P and T are the numbers of places and transitions) and in $O(P(P+T)^2)$ for live and bounded extended free-choice nets. Although these algorithms have a reasonably good computational complexity and are applicable to a large class of nets, large numbers of concurrent pairs of nodes may still lead to long computation times. This paper complements the palette of concurrency detection algorithms with the *Concurrent Paths* (CP) algorithm for sound (simple) free-choice workflow nets. The algorithm allows parallelization and has a worst-case computational complexity of $O(P^2 + T^2)$ for acyclic nets and of $O(P^3 + PT^2)$ for cyclic nets. Although the computational complexity of cyclic nets has not improved, the evaluation shows the benefits of CP, especially, if the net contains many nodes in concurrency relation.

Keywords

Concurrency detection, Workflow nets, Free-Choice, Soundness.

1. Introduction

Petri nets are a popular and well-studied notion to describe, investigate, revise, and analyze complex system behavior. Especially in the area of information systems and business process management, Petri nets are commonly used to model business processes. Instead of classical control-flow graphs resulting from procedural programming languages, nets are able to model concurrent behavior. Of course, although concurrency is an important aspect of nets (and the business process models they may represent), it complicates their analysis. The detection of concurrent places and transitions of nets helps to understand the behavior of a system. It further enables the measurement of similarity of nets with behavior [1], the derivation of other behavioral relations, such as e. g., causality, exclusivity [2, 3], etc. Some ongoing analysis, therefore, depends on concurrency detection and its efficiency.

Kovalyov [4] proposed a quintic time algorithm in the number of nodes that computes all concurrent places. Kovalyov and Esparza [5] revised that algorithm to obtain a time complexity of $O((P+T)TP^2)$ for live and bounded nets and of $O(P(P+T)^2)$ for live and bounded extended free-choice nets (P is the number of *places* and T is the number of *transitions* of a net). Both algorithms are very general, well-suited, and efficient for their respective problem classes of nets. However, if a net contains many concurrent nodes, the computation time increases significantly during evaluation.

Decomposition of the net can be used to allow a parallel computation of the concurrency relation and to accelerate the computational method. A decomposition into single-entry single-exit (SESE) fragments helped to speed-up the computation in Weidlich et al. [6] and Ha and Prinz [7]. However, the SESE

PNSE'24, International Workshop on Petri Nets and Software Engineering, 2024

*Corresponding author.

✉ Thomas.Prinz@uni-jena.de (T. M. Prinz); Julien.Klaus@uni-jena.de (J. Klaus); Nick.VanBeest@data61.csiro.au (N. R.T.P. v. Beest)

ORCID 0000-0001-9602-3482 (T. M. Prinz); 0000-0002-1498-2653 (J. Klaus); 0000-0003-3199-1604 (N. R.T.P. v. Beest)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).



decomposition approach fails if the net contains inherently unstructured fragments (known as rigids). Weber et al. [8] proposed a quadratic time variant to compute concurrent nodes. Their algorithm requires a low number of nodes in the pre- and postset as well as only simple cycles (loops). For other cases, SESE decomposition and the approach of Weber et al. have to utilize the algorithm of Kovalyov and Esparza or more general techniques, such as state-space exploration or finite complete prefix unfolding [9]. Therefore, these approaches are again at least cubic in time complexity or worse for nets with arbitrary loops.

To overcome these limitations, this paper presents a new algorithm, called the *Concurrent Paths* (CP) algorithm, which is applicable to *sound (simple) free-choice workflow nets*. This class of nets is more restricted than those for the Kovalyov and Esparza algorithms, since it requires an explicit start and end place, 1-boundedness, and the simpler definition of free-choiceness [10, 11]. For acyclic nets, however, it has a quadratic worst-case computation complexity of $O(P^2 + T^2)$ and, for cyclic nets, the worst-case complexity increases to a cubic algorithm of $O(P^3 + PT^2)$. The core idea of the approach is to consider concurrent paths instead of concurrent places and transitions. The CP algorithm is well parallelizable. In addition, the worst-case complexity occurs relatively infrequently, as it depends on the overall number of loops (incl. nested loops) in a net, which is usually small. If this number of loops can be interpreted as a constant, the complexity reduces to be quadratic in average. The restriction to connected workflow nets with single start and end places serves to introduce the method, but unconnected nets with several start and end places are also possible. In summary, the CP algorithm complements the palette of concurrency detection algorithms for the special case of sound (simple) free-choice workflow nets. A slightly extended version of this paper can be found as a technical report [12].

The remainder of the paper is structured as follows. Section 2 explains basic notions, especially nets, paths, loops, markings, and semantics. Subsequently, the concept of concurrency and the algorithm of Kovalyov and Esparza are introduced and described in Section 3. This is followed by revisions of their algorithm to the quadratic time CP algorithm for acyclic nets in Section 4. Section 5 extends the algorithm to the *Concurrent Paths* algorithm being able to handle cyclic nets. The evaluation in Section 6 demonstrates the strengths and weaknesses of the CP algorithm compared to the Kovalyov and Esparza algorithm for live and bounded free-choice nets. Finally, Section 7 summarizes the results and provides directions for future work.

2. Preliminaries

This work is based on well-known definitions of Petri nets. Readers already familiar with these concepts may proceed directly to Section 2.3. However, a quick review is encouraged to ensure alignment with this paper's specific notions.

2.1. Nets, Paths, Loops, and Workflow Nets

Definition 1 (Petri net). A *Petri net* is a triple $N = (P, T, F)$ with P and T are finite, disjoint sets of *places* and *transitions* and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*. J

The union $P \cup T$ of a net $N = (P, T, F)$ can be interpreted as *nodes* and F as *edges* between those nodes. For $x \in P \cup T$, $\bullet x = \{p \mid (p, x) \in F\}$ is the *preset* of x (all directly preceding nodes) and $x \bullet = \{s \mid (x, s) \in F\}$ is the *postset* of x (all directly succeeding nodes). Each node in $\bullet x$ is an *input* of x and each node in $x \bullet$ is an *output* of x . The preset and postset of a set of nodes $X \subseteq P \cup T$ is defined as $\bullet X = \bigcup_{x \in X} \bullet x$ and $X \bullet = \bigcup_{x \in X} x \bullet$, respectively.

Definition 2 (Path). A *path* $W = (n_1, \dots, n_m)$ of a net $N = (P, T, F)$ is a sequence of nodes $n_1, \dots, n_m \in P \cup T$ with $m \geq 1$ and $\forall i \in \{1, \dots, m-1\}: n_i \in \bullet n_{i+1}$. J

Of course, places and transitions alternate on paths. Nodes are part of a path $A = (x, \dots, y)$, depicted $x, \dots, y \in A$. If all nodes of a path are pairwise different, the path is *acyclic*; otherwise, it is *cyclic*. The acyclic path A is sometimes used as the set $\{x, \dots, y\}$. $Paths(x, y)$ denotes the set of all *acyclic*

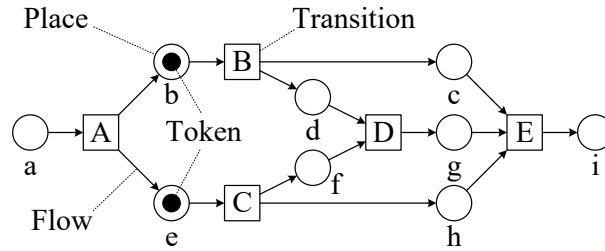


Figure 1: A graphical example of a Petri net.

paths between nodes x and y , where $x, y \in P \cup T$. Two paths $A = (x, \dots, y)$ and $B = (y, \dots, z)$ can be *concatenated* to a new path $A + B = (x, \dots, y) + (y, \dots, z) = (x, \dots, y, \dots, z)$. If A and B are acyclic and $A \cap B = \{y\}$, then $A + B$ is acyclic since all nodes of $A + B$ are pairwise disjoint. A net N is *cyclic* if there is at least one node $x \in P \cup T$ that has a non-trivial path ($\neq (x)$) to itself.

Cyclic nets contain at least one *loop*. A loop is a subgraph of the net in which each node is reachable from any other node, which can be formally defined as follows:

Definition 3 (Loop). A loop $L = (P_L, T_L, F_L)$ of a net $N = (P, T, F)$ is a *strongly connected component* of N , i. e., L is a maximal subgraph of N , such that $P_L \subseteq P$, $T_L \subseteq T$, and $F_L \subseteq F$ [13]. We denote the set of all loops of N with $Loops(N)$, such that $Loops(N) = \emptyset$ for acyclic nets. We further define $Entries(L)$ as the set $\{l \in (P_L \cup T_L) : \bullet l \not\subseteq (P_L \cup T_L)\}$ of *loop entries* of L , and $Exits(L)$ as the set $\{l \in (P_L \cup T_L) : l \bullet \not\subseteq (P_L \cup T_L)\}$ of *loop exits* of L . All flows in $\{(o, l) \in F : o \notin (P_L \cup T_L) \wedge l \in (P_L \cup T_L)\}$ are *loop-entry flows* and all flows in $\{(l, o) \in F : l \in (P_L \cup T_L) \wedge o \notin (P_L \cup T_L)\}$ are *loop-exit flows* of L .

Each net in this paper is restricted to be (*simple*) *free-choice*: $\forall p \in P : |p \bullet| > 1 \implies \bullet(p \bullet) = \{p\}$ [10, 11]. Note that the free-choice property is usually used in its *extended* version: $\forall t_1, t_2 \in T : \bullet t_1 \cap \bullet t_2 \neq \emptyset \implies \bullet t_1 = \bullet t_2$ [14]. Polyvyanyy [10] and Favre et al. [11] provide bijective translations from nets belonging to the extended definition into nets applying to the simpler free-choice property, through the use of silent transitions. In the following, we use the terms *free-choice* to refer to the simpler and *extended free-choice* to refer to the extended definition.

Visualized nets have circles representing places, rectangles representing transitions, and directed edges representing flows (see Figure 1).

Definition 4 (Workflow and AFW-net). A *workflow net* $WN = (P, T, F, i, o)$ is a net (P, T, F) with $i, o \in P$, $\bullet i = \emptyset$, and $o \bullet = \emptyset$. i is the *source* and o is the *sink* of WN . All nodes are on a path from i to o . If WN is free-choice, we call it *FW-net*. If WN is acyclic and free-choice, we call it *AFW-net*.

The visualized net in Figure 1 is a workflow (FW-)net as well as an AFW-net.

2.2. Markings, Semantics, Reachability, and Soundness

The state of a net is described by a so-called *marking*, which specifies the number of *tokens* in each place.

Definition 5 (Marking). A *marking* of a net $N = (P, T, F)$ is a total mapping $M : P \mapsto \mathbb{N}_0$ that assigns a natural number (inclusively 0) of *tokens* to each place P . $M(p) = 1$ means that place $p \in P$ carries 1 token in marking M . The mapping M is sometimes used as the set $\{p \in P : M(p) \geq 1\}$ representing only places having at least one token in M .

An *initial* marking of a net is a special marking with at least one place having a token. The initial marking M_i of a workflow net $WN = (P, T, F, i, o)$ is a marking $\{i\}$ where only the source i of a net contains exactly one token. A *terminal* marking M_o is a marking $\{o\}$ of WN where only the sink o contains exactly one token. Transitions whose input places all contain at least one token are *enabled* in a marking and can be fired. This leads to the execution semantics of a net:

Definition 6 (Execution semantics). Let $N = (P, T, F)$ be a net with a marking M . A transition $t \in T$ is *enabled* in M iff every input place of t contains at least one token, $\forall p \in \bullet t: M(p) \geq 1$. If t is enabled in M , then t can *occur* (“fire”), which leads to a *step* from M to M' via t , denoted $M \xrightarrow{t} M'$, with

$$\forall p \in P: M'(p) = M(p) - \begin{cases} 1, & p \in \bullet t \\ 0, & \text{else} \end{cases} + \begin{cases} 1, & p \in t \bullet \\ 0, & \text{else.} \end{cases} \quad (1)$$

I. e., in a step via t , t “consumes” one token from all its input places and “produces” one token for all its output places [2].

Stepwise firings of transitions lead to chains of fired transitions, which describe the behavior of a net as occurrence sequences:

Definition 7 (Occurrence Sequences and Reachability). Let $N = (P, T, F)$ be a net with a marking M_0 . A sequence of transitions $\sigma = \langle t_1, \dots, t_n \rangle$, $n \in \mathbb{N}_0$, $t_1, \dots, t_n \in T$, is an *occurrence sequence* of M_0 iff there is a sequence of markings M_0, M_1, \dots, M_n such that $M_{i-1} \xrightarrow{t_i} M_i$ holds for each $i \in \{1, \dots, n\}$. It can be said that σ *leads* from M_0 to M_n .

A marking M' is *reachable* from a marking M (denoted $M \rightarrow^* M'$) iff there is an occurrence sequence σ of M that leads to M' or $M = M'$.

Important properties of nets are *liveness* and *boundedness*:

Definition 8 (Liveness and Boundedness). A net $N = (P, T, F)$ with its initial marking M_0 is *live* iff for every reachable marking M , $M_0 \rightarrow^* M$, and every $t \in T$, there is a reachable marking M' , $M \rightarrow^* M'$, which enables t .

N is *n-bounded* iff there exists a number $n \in \mathbb{N}_0$ such that for every reachable marking M , $M_0 \rightarrow^* M$, and for every place $p \in P$ it holds that the number of tokens at p is at most n : $\forall p \in P: M(p) \leq n$. N is *safe* iff it is 1-bounded.

Soundness describes an important property of workflow nets:

Definition 9 (Soundness). A workflow net $WN = (P, T, F, i, o)$ with its initial marking $M_i = \{i\}$ and its terminal marking $M_o = \{o\}$ is *sound* iff

- (1) $\forall M, M_i \rightarrow^* M: M \rightarrow^* M_o$,
- (2) $\forall M, M_i \rightarrow^* M: (M(o) \geq 1 \implies M = M_o)$, and
- (3) there is no *dead* transition in WN : $\forall t \in T \exists M, M': M_i \rightarrow^* M \xrightarrow{t} M'$. [15]

Theorem 1. Let $WN = (P, T, F, i, o)$ be a FW-net with its initial marking $M_i = \{i\}$ and its terminal marking $M_o = \{o\}$ and $\overline{WN} = (P, T \cup \{t'\}, F \cup \{(o, t'), (t', i)\})$ be its “short-circuit” net.

$$WN \text{ is sound} \iff \overline{WN} \text{ is live and bounded}$$

Proof. See [15]. □

2.3. Path-to-End Theorem

To simplify proofs in the remainder of this paper, we use the *Path-to-End Theorem*. The theorem states that in sound FW-nets, there is no reachable marking, in which at least two tokens are on an acyclic path to the sink.

Theorem 2 (Path-to-End Theorem). Let $WN = (P, T, F, i, o)$ be a FW-net with its initial marking $M_i = \{i\}$ and its terminal marking $M_o = \{o\}$.

$$WN \text{ is sound} \implies \forall p \in P \forall W \in \text{Paths}(p, o) \forall M, M_i \rightarrow^* M: \sum_{w \in W \cap P} M(w) \leq 1 \quad (2)$$

Proof. The preconditions by Theorem 2 are a FW-net $WN = (P, T, F, i, o)$ with its initial marking $M_i = \{i\}$ and its terminal marking $M_o = \{o\}$. The proof is done by contradiction. Let us assume that:

$$\exists p \in P \exists W \in Paths(p, o) \exists M, M_i \rightarrow^* M: \sum_{w \in W \cap P} M(w) > 1 \quad \wedge \quad WN \text{ is sound} \quad (3)$$

By (3), let $p \in P$ be a place with an acyclic path $W \in Paths(p, o)$ to o and M a reachable marking from M_i , $M_i \rightarrow^* M$, such that $\sum_{w \in W \cap P} M(w) > 1$.

First of all, let us consider each transition $t \in W \cap T$ to check how much input places of each t can lay on path W . Assume, there is a transition $t \in W \cap T$ with more than one input place on W (i. e., $\exists t \in W \cap T: |\bullet t \cap W| > 1$). There are exactly two cases for such a $t \in W \cap T$ with $|\bullet t \cap W| > 1$:

- (1) $\exists w \in \bullet t \cap W: |w\bullet| > 1$ (at least one input place of t on path W has more than one output transition).
Following that path W is acyclic, w does not appear twice on W . As a consequence, there is at least one other input place w' in $\bullet t \cap W$, $w \neq w'$, and, thus, $|w\bullet| > 1$ and $\{w, w'\} \subseteq \bullet(w\bullet) = \bullet t$. Therefore, the FW-net WN is not simple free-choice. ζ
- (2) $\forall w \in \bullet t \cap W: |w\bullet| = |\{t\}| = 1$ (all input places of t on path W have exactly t as output transition).
Let $w_1, w_2 \in \bullet t \cap W$, $w_1 \neq w_2$, be such two input places of t on path W with $w_1\bullet = w_2\bullet = \{t\}$. Since the sink has an empty postset, $o\bullet = \emptyset$, w_1 and w_2 cannot be o , $w_1 \neq w_2 \neq o$. By Def. 2 of paths, each node $n \in W$ (except the last node o on path W) is directly followed by a node $n' \in W \cap n\bullet$ of n 's postset. For this reason and since $w_1 \neq w_2$, $w_1, w_2 \in W$, and $w_1\bullet = w_2\bullet = \{t\}$, w_1 and w_2 must directly be followed by t on path W . As a consequence, t appears at least twice on path W and, therefore, W 's nodes are not pairwise different. This implies that path W is cyclic and not acyclic. ζ

Both cases fail. As a result, each transition t on W has exactly one input place on W , $\forall t \in W \cap T: |\bullet t \cap W| = 1$. In general, it is valid for W :

$$\forall t \in (W \cap T): |\bullet t \cap W| = 1 \quad \wedge \quad |t\bullet \cap W| \geq 1 \quad (4)$$

As a consequence, firing a $t \in (W \cap T)$ in any marking M_1 cannot reduce the number of tokens on W :

$$\forall t \in (W \cap T) \forall M_1, M \rightarrow^* M_1, t \text{ is enabled in } M_1, \forall M_2, M_1 \xrightarrow{t} M_2: \sum_{w \in W \cap P} M_1(w) \leq \sum_{w \in W \cap P} M_2(w) \quad (5)$$

Thus, the only possibility to reduce the number of tokens on W is a place $p_w \in W \cap P$ with $|p_w\bullet| \geq 2$. Since $|o\bullet| = 0$, $p_w \neq o$. As a consequence:

$$\exists \{t, t'\} \subseteq p_w\bullet: t, t' \in T \quad \wedge \quad t \notin W \quad \wedge \quad t' \in W \quad (6)$$

Let t, t' be such transitions. Since WN is simple free-choice, it is valid for p_w that $\bullet(p_w\bullet) = \{p_w\bullet\} = \bullet t \cup \bullet t'$ and, therefore:

$$\bullet t = \bullet t' = \{p_w\bullet\} \quad (7)$$

It results from (7) and for each place p_w on W with $|p_w\bullet| \geq 2$ that each of p_w 's output transition t is enabled in each reachable marking M_1 from M , in which p_w has a token:

$$\forall p_w \in W \cap P, |p_w\bullet| \geq 2 \forall t \in p_w\bullet \forall M_1, M \rightarrow^* M_1, M_1(p_w) \geq 1: t \text{ is enabled in } M_1 \quad (8)$$

We construct an occurrence sequence starting from M with a simple rule: If a $p \in (W \cap P)$ with $|p\bullet| \geq 2$ has a token in a reachable marking M_1 , $M \rightarrow^* M_1$, then $t \in (p\bullet \cap W)$ fires following (8). We depict a reachable marking under this rule with \rightarrow^R instead of \rightarrow^* , i. e., $M \rightarrow^R M_1$. For this reason, following (5) and following this rule:

$$\forall M_1, M \rightarrow^R M_1: \sum_{w \in W \cap P} M_1(w) > 1 \quad (9)$$

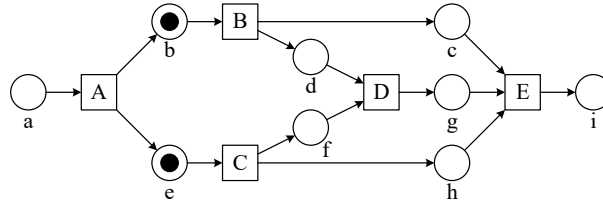


Figure 2: A net with concurrency. It holds that $(b, h) \in \parallel$ but $(b, g) \notin \parallel$.

Since WN is sound by (3) and W is finite, there is at least marking M_o , in which no transition is enabled. Therefore, let M_1 be a reachable marking under this rule, in which no transition is enabled:

$$M \rightarrow^R M_1 \quad \forall t \in T: t \text{ not enabled in } M_1 \quad (10)$$

Note, it is *not* possible that a transition in (10) cannot be enabled because of the applied rule. Since WN is sound by (3), the marking M_1 by (10) cannot be the terminal marking M_o by Def. 9 since $\sum_{w \in W \cap P} M_1(w) > 1$ by (9). Therefore, there must be at least one *dead* transition on W , which contradicts soundness by Def. 9. This violates contradiction (3), proving the validity of Theorem 2. \checkmark \square

3. Concurrency and the Algorithm of Kovalyov and Esparza

A node $x \in P \cup T$ is called *active* in a marking M , depicted as x_M^* , in the following if it either contains a token ($x \in P$ and $M(x) \geq 1$) or is enabled in M ($x \in T, \forall p \in \bullet x: M(p) \geq 1$).

Definition 10 (Concurrency). For a given sound (simple) free-choice workflow net $WN = (P, T, F, i, o)$ and its initial marking $M_i = \{i\}$, there is a *concurrency relation* $\parallel \subseteq (P \cup T) \times (P \cup T)$ with $(x, y) \in \parallel$, $x \neq y$, iff there is a reachable marking M , $M_i \rightarrow^* M$, with:

- (1) x_M^* and y_M^* , and
- (2) if $x \in T$ and $y \in P$, then $\forall z \in \bullet x: (z, y) \in \parallel$, and
- (3) if $x \in P$ and $y \in T$, then $\forall z \in \bullet y: (z, x) \in \parallel$, and
- (4) if $x \in T$ and $y \in T$, then $\forall z \in \bullet x \forall z' \in \bullet y: (z, z') \in \parallel$.

\parallel is symmetric [5]. Following Verbeek et al. [16], sound free-choice workflow nets are safe (i. e., 1-bounded). As a consequence, \parallel is irreflexive (i. e., a place/transition cannot be concurrent to itself). \lrcorner

The cases (2)–(4) exclude that a place is in concurrency relation with its output transitions and that two transitions are in concurrency relation, which require the same tokens from shared input places. For example, places e and d of Figure 1 are in a concurrency relation, $(e, d) \in \parallel$. Both places can be active (i. e., have a token) in a reachable marking. Place d and transition C are also in a concurrency relation since there is a reachable marking in which d has a token, C is enabled, and the input place e is in a concurrency relation with d (cases (2)–(4)). $(f, D) \notin \parallel$ although f and D can be active in a reachable marking. However, following cases (2)–(3), f must be in a concurrency relation with itself, which is not possible since \parallel is irreflexive.

Kovalyov and Esparza [5] defined a cubic $O(|P|(|P| + |T|)^2)$ algorithm to identify the \parallel relation for live, bounded, and extended free-choice nets. We refer to the algorithm as the *KovEs* algorithm. The *KovEs* algorithm is applicable to a wide range of net classes and is, therefore, more general than the algorithm provided in this paper. In the remainder of this section, we provide an overview of its most important concepts and explain its functionality. In doing this, we focus on *KovEs* in the context of sound, simple free-choice workflow nets to revise the algorithm regarding this class of nets.

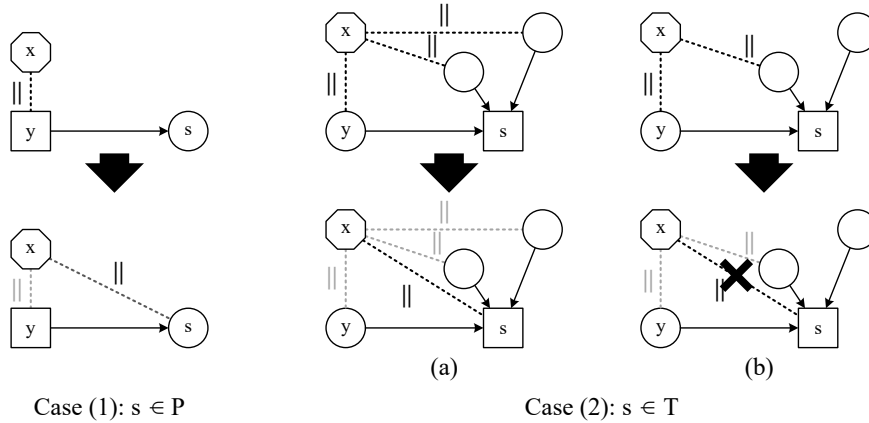


Figure 3: For a pair $(x,y) \in \parallel$, there are two cases of candidate for (x,s) , $s \in y\bullet$: $s \in P$ or $s \in T$. x is visualized as an octagon since it could be either a place or transition. The dotted lines highlight nodes being in a concurrency relation. Arcs denote ordinary flow relations. Above the big black arrows, there are the situations *before* new pairs in relation were derived. Below the black arrows, there are situations with newly derived relations (in black). Note that in Case (2) (b) the relation cannot be derived.

The initial step of the *KovEs* algorithm is based on a simple observation in free-choice workflow nets $WN = (P, T, F, i, o)$ with its initial marking $M_0 = \{i\}$: all places in the postset of transitions are pairwise in a concurrency relation (except for all reflexive pairs):

$$\forall t \in T \forall x, y \in t\bullet, x \neq y: (x, y) \in \parallel \quad ((y, x) \in \parallel) \quad (11)$$

For example, in Figure 2, the pairs $\{(b, e), (e, b), (c, d), (d, c), (f, h), (h, f)\} \subseteq \parallel$. Besides transitions, all *source places* containing tokens in M_0 can be handled in the same way:

$$\forall x, y \in P, x \neq y, M_0(x) \geq 1 \leq M_0(y): (x, y) \in \parallel \quad ((y, x) \in \parallel) \quad (12)$$

The *KovEs* algorithm extends an initial set of the concurrency relation R by considering each already detected pair $(x, y) \in \parallel$. Since $(x, y) \in \parallel$, x (y) may also be concurrent to nodes in $y\bullet$ ($x\bullet$). For example, (b, C) is a new candidate for (b, e) in Figure 2. For x , this means that $\{(x, s) : s \in y\bullet\}$ are new candidate pairs to be concurrent. For each candidate (x, s) , there are exactly two cases: **(1)** s is a place ($s \in P$ and $y \in T$), or **(2)** s is a transition ($s \in T$ and $y \in P$). Figure 3 visualizes both cases, where node x is visualized as an octagon since it could be either a place or transition.

Consider case **(1)**: Following from Def. 10 of concurrency, there is a reachable marking M from M_0 , $M_0 \rightarrow^* M$, in which x and y are active, thus, y is enabled. In a step where y fires in M , $M \xrightarrow{y} M'$, s contains a token in the resulting marking M' and x is still active in M' : x and s are active in M' . For this reason, for case **(1)**, all nodes in $y\bullet$ are concurrent to x , leading to the following observation already used in [5]:

Observation 1. Let $WN = (P, T, F, i, o)$ be a sound free-choice workflow net.

$$\forall x \in (P \cup T) \forall y \in T: \\ (x, y) \in \parallel \iff \forall s \in y\bullet: (x, s) \in \parallel$$

The left side of Figure 3 visualizes concurrency relations with dotted lines.

Considering case **(2)** ($s \in T$ and $y \in P$), there is a reachable marking M from M_0 , $M_0 \rightarrow^* M$, in which x and y are active (i. e., $M(y) \geq 1$). Following [5], $(x, s) \in \parallel$ if and only if x is concurrent to $\bullet s$. Otherwise, s and x cannot be active in a reachable marking from M . Case **(2)** (a) of Figure 3 illustrates such a situation, in which x is concurrent to $\bullet s$ and also to s , $(x, s) \in \parallel$. Case **(2)** (b) of Figure 3 visualizes a situation, in which x is not concurrent to the upper right place but for the other places, such that $(x, s) \notin \parallel$. In general, it results in the following observation:

Observation 2. Let $WN = (P, T, F, i, o)$ be a sound free-choice workflow net.

$$\forall x \in (P \cup T) \forall y \in P \forall s \in y \bullet : \\ (\forall p \in \bullet s : (x, y) \in \parallel \wedge (x, p) \in \parallel) \implies (x, s) \in \parallel$$

Algorithm 1 depicts the resulting algorithm of Kovalyov and Esparza [5]. It differs in its notation from the original in four ways: At first, it explicitly ignores reflexive pairs of nodes $(x, x) \in (P \cup T) \times (P \cup T)$; second, it uses the union-operator \cup for generating pairs to allow us a simpler discussion about its time complexity; third, line 9 is simplified to just $E \leftarrow R$ instead of $E \leftarrow R \cap ((P \cup T) \times P)$ from the original paper since all pairs in R must fulfill the condition by lines 3 and 5; and, last, it uses the relation A as a mapping in lines 7 and 14 instead of a set.

Readers, who are not familiar with the algorithm, will find an explanation of the algorithm in the following paragraph. Familiar readers may proceed directly to Section 4. In Algorithm 1, lines 3 and 5 determine the initial concurrency relations discussed as “initial step” above. Line 7 defines a mapping A from each place to its directly succeeding places (i. e., the union of postsets of the postset of a place). This is used in line 14 to quickly determine new pairs in relation. E in line 9 is a set that contains new pairs not investigated yet. The while loop (lines 10–16) is iterated until E is empty. It takes an arbitrary pair (x, p) out of E (line 11) and one random transition t of the postset of p (line 12). Following case (2) discussed above, x is in concurrency relation with each place in the postset of t (lines 14–16) if it is already in relation with each place in $\bullet t$ (line 13). E is extended with all new pairs (line 15) and R is extended as well (line 16). Although Kovalyov and Esparza state that their algorithm identifies all pairs of nodes in concurrency relation, it “only” identifies pairs of *places* in relation. For this reason, case (1) discussed above is just indirectly important in Algorithm 1 but will be used for revisions in the next section.

Algorithm 1 The Kovalyov and Esparza algorithm [5] to determine \parallel for a given sound free-choice workflow net $WN = (P, T, F, i, o)$ with its initial marking M_0 .

```

1: function DETERMINEKOVESCONCURRENCY( $WN = (P, T, F, i, o), M_0$ )
2:   // Add pairs of places carrying a token in  $M_0$ .
3:    $R \leftarrow \bigcup_{p_1 \in \text{Src}(N), M_0(p_1) \geq 1} \bigcup_{p_2 \neq p_1 \in \text{Src}(N), M_0(p_2) \geq 1} \{(p_1, p_2)\}$ 
4:   // Add pairs of places of the postset of each transition.
5:    $R \leftarrow R \cup \bigcup_{t \in T} \bigcup_{p_1 \in t \bullet} \bigcup_{p_2 \neq p_1 \in t \bullet} \{(p_1, p_2)\}$ 
6:   // Determine for each place  $p$  its post-postset.
7:    $A \leftarrow \bigcup_{p \in P} \bigcup_{p' \in (p \bullet) \bullet} \{(p, p')\}$ 
8:   // Initialize the set of new relations.
9:    $E \leftarrow R$ 
10:  while  $E \neq \emptyset$  do
11:    Take  $(x, p) \in E$  and remove it  $E \leftarrow E \setminus \{(x, p)\}$ .
12:    Take  $t \in p \bullet$ .
13:    if  $\bigcup_{y \in \bullet t} \{(x, y)\} \subseteq R$  then
14:       $tmp \leftarrow \bigcup_{p' \in A(p)} \{(x, p')\}$ 
15:       $E \leftarrow E \cup (tmp \setminus R)$ 
16:       $R \leftarrow R \cup tmp$ 
17:  return  $R$ 

```

4. Quadratic Algorithm for AFW-Nets

The *KovEs* algorithm has a worst-case time complexity of $O(|P|(|P| + |T|)^2)$ resulting from its wide range of net classes for which it is applicable. This worst case appears if nets have a high level of concurrency, leading to long computation times as the evaluation will show later. This section revises the *KovEs* algorithm for AFW-nets based on soundness, the *simpler* free-choice property, and Observations 1 and 2 discussed in Section 3. An extension for cyclic FW-nets is presented in the next section.

The core idea of the revisions is to explore paths instead of directly succeeding nodes. As a consequence of the Path-to-End Theorem 2, a place cannot be in a concurrency relation with all places on all paths from this place to the sink. Therefore, for two concurrent places only those successors can be in a concurrency relation for which there is no path in-between. This core idea is investigated and refined in the following.

Concurrency of two different transitions in sound AFW-nets requires the absence of any path between them:

Theorem 3. Let $WN = (P, T, F, i, o)$ be a sound AFW-net with its initial marking $M_i = \{i\}$ and its terminal marking $M_o = \{o\}$.

$$\forall x, y \in P \cup T, x \neq y: \\ (\text{Paths}(x, y) \cup \text{Paths}(y, x)) \neq \emptyset \implies (x, y) \notin \parallel$$

Proof. The precondition by Theorem 3 is a sound AFW-net $WN = (P, T, F, i, o)$ with its initial marking $M_i = \{i\}$ and its terminal marking $M_o = \{o\}$. The proof is done by contradiction. We assume there is a case, where x has a path to y or y has a path to x and both are still in a concurrency relation:

$$\forall x, y \in P \cup T, x \neq y: (\text{Paths}(x, y) \cup \text{Paths}(y, x)) \neq \emptyset \quad \wedge \quad (x, y) \in \parallel \quad (13)$$

Without loss of generality and following from this contradiction (13), let x and y be two nodes with $(x, y) \in \parallel$ and there is a path $A_{x \rightarrow y}$ from x to y :

$$x, y \in P \cup T, x \neq y, (x, y) \in \parallel \quad \wedge \quad \text{Paths}(x, y) \neq \emptyset \quad \wedge \quad A_{x \rightarrow y} \in \text{Paths}(x, y) \quad (14)$$

Following from Def. 10 of concurrency, there is a marking M where x and y are active:

$$\exists M, M_i \rightarrow^* M: \overset{*}{x}_M \wedge \overset{*}{y}_M \quad (15)$$

To simplify ongoing considerations and following from (14) and (15), we pick two *places* x' and y' carrying a token in M . If x (y) is a place, then $x' = x$ ($y' = y$); otherwise, x' (y') is a place in $\bullet x$ ($\bullet y$):

$$x' = \begin{cases} x, & x \in P \\ p \in (\bullet x \cap M), & x \in T \end{cases}, \quad y' = \begin{cases} y, & y \in P \\ p \in (\bullet y \cap M), & y \in T \end{cases} \quad (16)$$

Since x' and y' are places and WN is sound, $M(x') = M(y') = 1$. Furthermore, let $A_{x' \rightarrow y'}$ be the modified path of $A_{x \rightarrow y}$ with adding x' or removing y if necessary. Following from Def. 4, there is a path $B_{y' \rightarrow o}$ from y' to the sink o of WN :

$$\text{Paths}(y', o) \neq \emptyset \quad \wedge \quad B_{y' \rightarrow o} \in \text{Paths}(y', o) \quad (17)$$

Since WN is acyclic by precondition:

$$A_{x' \rightarrow y'} + B_{y' \rightarrow o} = C \quad \wedge \quad C \text{ is acyclic} \quad (18)$$

As a consequence from (15), (17), and (18), the sum of all tokens on the acyclic path C in M is more than 2:

$$\sum_{c \in C \cap P} M(c) \geq |\{x', y'\}| = 2 \quad (19)$$

Since WN is sound, (19) contradicts the Path-to-End Theorem 2 $\frac{1}{2}$. As a consequence, WN cannot be sound and the contradiction fails. The theorem is valid. \checkmark \square

Following from Theorem 3 above, if two nodes are in concurrency relation, there is no path between them.

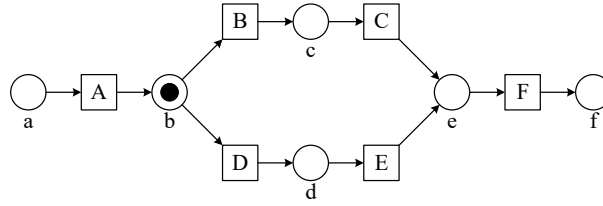


Figure 4: A simple net with exclusive paths, in which, for example place c and place d can never be concurrent.

Corollary 4. Let $WN = (P, T, F, i, o)$ be a sound AFW-net with its initial marking $M_i = \{i\}$ and its terminal marking $M_o = \{o\}$.

$$\forall x, y \in P \cup T, x \neq y: \\ (x, y) \in \parallel \implies Paths(x, y) = Paths(y, x) = \emptyset$$

Proof. The statement directly follows from the contraposition of Theorem 3. \square

For example, the AFW-net in Figure 2 has concurrent transitions B and C , $(B, C) \in \parallel$, and concurrent places b and e , $(b, e) \in \parallel$. After firing transition C , $M(f) \geq 1$ and $M(h) \geq 1$, such that $(b, h) \in \parallel$. As Cor. 4 states, there is neither a path from place b to e nor from place b to h . For place g , there is a path from place b to g ; following Theorem 3, $(b, g) \notin \parallel$. Regarding Cor. 4, each combination of two (different) concurrent *active* nodes cannot have any path in-between for sound AFW-nets.

Although the absence of paths between two nodes is a necessary condition for concurrency in AFW-nets, it is not sufficient. For example, in Figure 4, place c has no path to place d , but both are never concurrent.

4.1. First Revision

The existence or absence of paths between two nodes of an AFW-net is used in our approach to revise the *KovEs* algorithm. In doing this, we reconsider Observations 1 and 2 from Section 3. Recall that, given a pair of nodes $(x, y) \in \parallel$ and a node $s \in y^\bullet$, we need to determine whether $(x, s) \in \parallel$. Observation 1 with $y \in T$ states that $\forall s \in y^\bullet, (x, s) \in \parallel$. Following Cor. 4, there cannot be a path from x to s :

Corollary 5. Let $WN = (P, T, F, i, o)$ be a sound AFW-net with its initial marking $M_i = \{i\}$ and its terminal marking $M_o = \{o\}$.

$$\forall y \in T \forall (x, y) \in \parallel \forall s \in y^\bullet: \\ Paths(x, s) = \emptyset \iff (x, s) \in \parallel$$

Proof. Constructive proof for $y \in T$, $(x, y) \in \parallel$, $s \in y^\bullet$, and the initial marking $M_i = \{i\}$ and the terminal marking $M_o = \{o\}$. We prove both directions:

$Paths(x, s) = \emptyset \implies (x, s) \in \parallel$: By Cor. 4 and since $(x, y) \in \parallel$, there cannot be a path between x and y , $Paths(x, y) = \emptyset$. Therefore, $x \notin y^\bullet$.

Following Def. 10 of concurrency and since $(x, y) \in \parallel$, there is a marking M with x_M^* and y_M^* :

$$\exists M: M_i \rightarrow^* M \quad \wedge \quad x_M^* \quad \wedge \quad y_M^* \tag{20}$$

Since y is active in M and $y \in T$, y is enabled in M . As a consequence, firing y in M can lead to a marking M' by Def. 6, in which s is active:

$$\exists M': M \xrightarrow{y} M' \quad \wedge \quad x_{M'}^* \quad \wedge \quad s_{M'}^* \tag{21}$$

By Def. 10 of concurrency and since $Paths(x, s) = \emptyset$ and $Paths(y, x) = \emptyset$ by Cor. 4: $(x, s) \in \parallel$. \checkmark

$(x, s) \in \parallel \implies Paths(x, s) = \emptyset$: By Cor. 4 and since $(x, s) \in \parallel$: $Paths(x, s) = \emptyset$. \checkmark

□

Observation 2 with $y \in P$ states that x is only concurrent to a transition $s \in y\bullet$ if it is also concurrent to all places p in $\bullet s$. This statement is complex to check but can be simplified by considering paths:

Corollary 6. Let $WN = (P, T, F, i, o)$ be a sound AFW-net with its initial marking $M_i = \{i\}$ and its terminal marking $M_o = \{o\}$.

$$\forall y \in P \forall (x, y) \in \parallel \forall s \in y\bullet : \\ Paths(x, s) = \emptyset \iff (x, s) \in \parallel$$

Proof. Constructive proof for $(x, y) \in \parallel$, $y \in P$, $s \in y\bullet$, and the initial marking $M_i = \{i\}$ and the terminal marking $M_o = \{o\}$. Let

$$M, M_i \xrightarrow{*} M \quad \wedge \quad x_M^* \quad \wedge \quad y_M^* \tag{22}$$

We prove both directions:

$Paths(x, s) = \emptyset \implies (x, s) \in \parallel$: If x would have a path to any node in $\bullet s$, it would have a path to s . However, $Paths(x, s) = \emptyset$. For this reason, x cannot have a path to any node in $\bullet s$:

$$\forall p \in \bullet s : Paths(x, p) = \emptyset \tag{23}$$

As a consequence, since WN is sound, there must be a reachable marking M' , in which x is still active and s became enabled (active), i. e., firing s is “independent” from x , because there is no path from x to s by (23):

$$\exists M' : M \xrightarrow{*} M' \quad \wedge \quad x_{M'}^* \quad \wedge \quad s_{M'}^* \tag{24}$$

Following Def. 10 and since $Paths(x, s) = \emptyset$ and $Paths(y, x) = \emptyset$ by Cor. 4, $(x, s) \in \parallel$. \checkmark

$(x, s) \in \parallel \implies Paths(x, s) = \emptyset$: By Cor. 4 and since $(x, s) \in \parallel$: $Paths(x, s) = \emptyset$. \checkmark

□

In summary, for both cases $y \in P$ and $y \in T$, if and only if x has no path to s , then $(x, s) \in \parallel$. Otherwise, $(x, s) \notin \parallel$. This is summarized by the following theorem:

Theorem 7. Let $WN = (P, T, F, i, o)$ be a sound AFW-net with its initial marking $M_i = \{i\}$ and its terminal marking $M_o = \{o\}$.

$$\forall (x, y) \in \parallel \forall s \in y\bullet : \\ Paths(x, s) = \emptyset \iff (x, s) \in \parallel$$

Proof. The theorem combines all cases from Cor. 5 and Cor. 6. \checkmark

□

Although this can already be used to revise the *KovEs* algorithm, we consider an additional revision as well.

4.2. Second Revision

Following the first revision, paths play a crucial role to identify concurrency. For this reason, we define an auxiliary relation:

Definition 11 (*HasPath* relation). Let $WN = (P, T, F, i, o)$ be an AFW-net. The *HasPath* relation

$$HasPath = \{(x, y) \in (P \cup T) \times (P \cup T) \mid Paths(x, y) \neq \emptyset\} \tag{25}$$

specifies whether a node x has an acyclic path to node y (*HasPath* is reflexive, so $(x, x) \in HasPath$). $HasPath(x)$ denotes the set of all nodes to which x has a path (again, inclusive of itself).

□

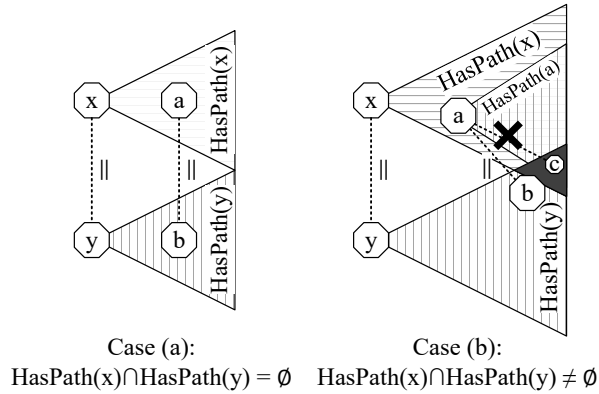


Figure 5: Two cases for mappings $\text{HasPath}(x)$ and $\text{HasPath}(y)$ of a concurrent pair $(x, y) \in \parallel$.

In the following, let $(x, y) \in \parallel$ for a sound AFW-net. From $(x, y) \in \parallel$ and Cor. 4 it follows that $y \notin \text{HasPath}(x)$ and $x \notin \text{HasPath}(y)$. Let us consider any node $a \in \text{HasPath}(x)$ to which x has a path. If a would have a path to y (i. e., $y \in \text{HasPath}(a)$), then x would have a path to y via a . For this reason, no node $a \in \text{HasPath}(x)$ has a path to y and no node in $\text{HasPath}(y)$ has a path to x .

Regarding $(x, y) \in \parallel$, there are exactly two cases: **(a)** No path starting in x to any sink crosses any path starting in y to any sink (i. e., $\text{HasPath}(x) \cap \text{HasPath}(y) = \emptyset$, cf. Figure 5 Case (a)); or **(b)** at least one path starting in x to a sink crosses at least one path starting in y to a sink (i. e., $\text{HasPath}(x) \cap \text{HasPath}(y) \neq \emptyset$, cf. Figure 5 Case (b)).

To case **(a)**: each node in $\text{HasPath}(x)$ must be in concurrency relation with any node in $\text{HasPath}(y)$ after a stepwise application of Theorem 7 regarding the first revision (and, since tokens in both sets can never converge). This is illustrated in Figure 5 by nodes a and b being concurrent.

To case **(b)**: the nodes of $\text{HasPath}(x)$ are partially overlapping with the nodes of $\text{HasPath}(y)$. This is illustrated in Figure 5 Case (b) with a gray triangle subset. Let $R_x^y = \text{HasPath}(x) \setminus \text{HasPath}(y)$ be the subset of nodes of $\text{HasPath}(x)$ to which y has no path (and, therefore, no node in $\text{HasPath}(y)$ as explained previously). A node $a \in R_x^y$ has a subset $\text{HasPath}(a) \subseteq \text{HasPath}(x)$ (illustrated as the grid triangle subset in Figure 5). Although no node in $\text{HasPath}(y)$ has a path to a , a may have paths to nodes in $\text{HasPath}(y)$ (more precisely, to nodes in $\text{HasPath}(a) \cap \text{HasPath}(y)$; in Figure 5 this is the subset of the gray triangle intersecting the grid triangle). Regarding Theorem 3, a cannot be concurrent to those nodes (c in the illustration). $R_y^a = \text{HasPath}(y) \setminus \text{HasPath}(a)$ contains all nodes of $\text{HasPath}(y)$ to which a has no path (and so no other node of $\text{HasPath}(a)$). As a consequence, for each node $a \in R_x^y$, there are sets $\text{HasPath}(a)$ and $\text{HasPath}(y) \setminus \text{HasPath}(a) = R_y^a$ being disjoint. There are disjoint paths from x to each node in $\text{HasPath}(a)$ and from y to each node in R_y^a . Following the first revision and Theorem 7, a step-wise consideration of these paths leads to concurrency between all nodes in $\text{HasPath}(a)$ and R_y^a . Instead of considering these nodes step-by-step, their pairwise concurrency can be added directly, i. e., $\text{HasPath}(a) \times R_y^a \subseteq \parallel$. This leads to a revised algorithm for sound AFW-nets with a quadratic computational complexity.

4.3. Revised Algorithm

Algorithm 2 defines the revised algorithm for sound AFW-nets. The algorithm computes the concurrency relation for each node of the net. However, if necessary, it can be modified to just compute the relations for places. Furthermore, the algorithm assumes the relation \parallel to be represented as an *adjacency list*. Naturally, it takes just $O(\parallel \parallel)$ to put it into a set of pairs.

Lines 2–6 of Algorithm 2 initialize the algorithm. Lines 3–5 initialize the set R for each unprocessed node with an empty set (following the *KovEs* algorithm, which is here the adjacency list of concurrent nodes). This initialization is linear to the number of nodes, $O(|P| + |T|)$. Line 6 computes the HasPath relation as adjacency list by calling a function *computeHasPath* of Algorithm 3. The algorithm computes HasPath utilizing a reverse topological order (each node appears *after* all its output nodes). Such order

Algorithm 2 The acyclic *Concurrent Paths* algorithm to determine \parallel for a given sound AFW-net $WN = (P, T, F, i, o)$.

```

1: function DETERMINECONCURRENCY( $WN = (P, T, F, i, o)$ ,  $R = \emptyset$ )
2:   // Initialize
3:   for all  $x \in P \cup T$  do
4:     if  $x \notin R$  then
5:        $R(x) \leftarrow \emptyset$ 
6:    $HasPath \leftarrow \text{COMPUTEHASPATH}(WN)$ 
7:   // Compute
8:    $I \leftarrow \emptyset$ 
9:   for all  $t \in T$  do
10:    for all  $x \in t \bullet$  do
11:       $I(x) \leftarrow I(x) \cup (t \bullet \setminus \{x\})$ 
12:    for all keys  $x \in I$  do
13:      for all  $y \in I(x)$  do
14:         $R_x^y \leftarrow HasPath(x) \setminus HasPath(y)$ 
15:        for all  $a \in R_x^y$  do
16:           $R(a) \leftarrow R(a) \cup (HasPath(y) \setminus HasPath(a))$ 
17:   return  $R$ 

```

Algorithm 3 Computation of *HasPath* as adjacency list for a given AFW-net $WN = (P, T, F, i, o)$.

```

1: function COMPUTEHASPATH( $WN = (P, T, F, i, o)$ )
2:   // Initialize
3:    $HasPath \leftarrow \emptyset$ 
4:   for all  $x \in P \cup T$  do
5:      $HasPath(x) \leftarrow \emptyset$ 
6:    $L \leftarrow P \cup T$  in reverse topological order starting from  $o$ 
7:   for all  $x \in L$  do
8:      $HasPath(x) \leftarrow \{x\} \cup \bigcup_{s \in x \bullet} HasPath(s)$ 
9:   return  $HasPath$ 

```

can be computed in $O(|P| + |T| + |F|)$ [13]. Thus, Algorithm 3 can compute *HasPath* for all nodes in $O(|P| + |T| + |F|)$ as well since each node ($|P \cup T|$) is investigated with its outputs ($|F|$). Assuming that set operations (such as \cup) can be achieved in constant time, e. g., with a *BitSet*, *HasPath* can be computed for all nodes in $O(|P| + |T| + |F|)$.

The actual computation of the concurrency relation takes place in lines 8–17. It creates a new adjacency list I for initial places in concurrency relation (line 8). It investigates all transitions t (line 9) and its output places x (line 10). Each x is initial concurrent to all other output places of t (line 11). Lines 12–13 investigate all pairs of different places being initially in concurrency relation in I . Line 14 computes the set R_x^y containing all nodes to which x but not y has a path. As discussed in the *Second Revision*, each node $a \in R_x^y$ is investigated in lines 15–16, where line 16 adds all nodes in $HasPath(y) \setminus HasPath(a)$ to be concurrent to a (i. e., all nodes in $HasPath(y)$ to which a does not have a path; recall that a does not have a path to any node of $HasPath(y)$ because it is in R_x^y).

The time complexity for lines 8–16 seems biquadratic, $O(X^4)$, at the first view. However, let us change the perspective on the algorithm focusing on line 16. If line 16 *adds* new information to $R(a)$, at least one new pair in concurrency was detected. That means if line 16 *always* adds new information $R(a)$ and, therefore, finds a new pair in concurrency, lines 8–16 can execute line 16 *at most* $\parallel \leq |(P \cup T) \times (P \cup T)|$ times, i. e., $O(|P|^2 + |T|^2)$ — the algorithm would be quadratic. In other words, line 16 is never unnecessarily executed, as we will explain in the following.

The main task of lines 8–13 is to create and consider a *new* pair (x, y) , which is taken from each $t \bullet$ of t in lines 8–11. There can only be $|P|^2$ of such pairs (as adjacency list) in I . Apart from this pair (and, of course, (y, x)), let us consider all other “initial” pairs $(\alpha, \beta) \neq (x, y) \neq (y, x)$ of lines 8–11. If for transition t with $x, y \in t \bullet$ it is valid that $t \in \text{HasPath}(\alpha)$, then $t \bullet \subset \text{HasPath}(\alpha)$, and if it is valid that $t \in \text{HasPath}(\beta)$, then $t \bullet \subset \text{HasPath}(\beta)$. For this reason, it is valid that either:

1. $\{x, y\} \subset \text{HasPath}(\alpha)$,
2. $\{x, y\} \subset \text{HasPath}(\beta)$, or
3. $\{x, y\} \not\subset (\text{HasPath}(\alpha) \cup \text{HasPath}(\beta))$.

As a consequence, all “initial” pairs (x, y) and (y, x) cannot be added by the handling of other “initial” pairs. The same holds true for all nodes in R_x^y of line 14 whose information to be concurrent to y is firstly added in line 16. Line 16, therefore, *always* adds new information. As explained earlier, line 16 can at most be executed in quadratic complexity making lines 8–16 quadratic, $O(|P|^2 + |T|^2)$.

In summary, assuming that set operations are applicable in constant time (like \cup and \setminus), e. g., by a *BitSet*, Algorithm 2 can be computed in $O((|P| + |T|) + (|P| + |T| + |F|) + |P| + |P|^2 + |T|^2) = O(3|P| + 2|T| + |F| + |P|^2 + |T|^2) = O(|P|^2 + |T|^2)$ in the worst-case (as $|F|$ is quadratic to $|P| + |T|$ in the worst-case). Since at least those pairs in concurrency must be investigated, which are a subset of $(P \cup T) \times (P \cup T)$, it will be difficult to obtain a faster algorithm in terms of asymptotic time complexity than $O(|P|^2 + |T|^2)$.

5. Cubic Algorithm for FW-Nets

Although AFW-nets are not unusual, concurrency in cyclic sound FW-nets must be considered as well. In contrast to the *KovEs* algorithm, Algorithm 2 cannot be applied to cyclic nets since the revisions are applicable to acyclic nets only. To overcome this situation, we use a method called *loop decomposition* [17] to decompose a cyclic sound FW-net into a set of sound AFW-nets with the same behavior. Cyclic nets become acyclic by replacing loops with single *loop places*. Figure 6 shows a cyclic net on the left side, which is decomposed into four acyclic nets on the right side. Subsequently, we apply Algorithm 2 to each acyclic net and combine all collected concurrency information, i. e., all nodes concurrent to an inserted *loop place* are concurrent to all nodes in the corresponding loop.

5.1. Loop Decomposition

The method of loop decomposition [17] was introduced to decompose sound *workflow graphs* with loops into sets of sound workflow graphs without loops. This decomposition method was slightly revised in [18]. Favre et al. [11] have shown how free-choice nets can be transferred into workflow graphs and vice versa. The notion of soundness used in the work of loop decomposition is equal to soundness of FW-nets [19]. For this reason, loop decomposition can be applied to sound FW-nets without strong modifications. Algorithm 4 describes the algorithm abstractly. The following only focuses on the consequences for concurrency detection. For further details about loop decomposition, we refer to previous work [17, 18].

Loop decomposition identifies loops as strongly connected components (cf. Def. 3 of loops). Following [17], entries and exits of loops are places in sound FW-nets. The (sometimes unconnected) subgraph between all loop entries and those loop exits being reachable without passing another loop exit is called the *do-body* of the loop. Once a loop exit contains a token, no other place in the loop or the do-body contains a token [17]. Therefore, the do-body can be interpreted as an implicit, initial “converging area” of different concurrent tokens before first loop exits are reached. Loop decomposition duplicates the do-body as an explicit, initial converging area before the loop. Subsequently, it replaces the entire loop (without the copied do-body) with a single *loop place* representing the previous loop. All flows into and out of the loop are redirected to start from and end at the loop place. Repeating this procedure with any (nested) loop finally leads to an acyclic sound FW-net [17]. For example, the cyclic network on the left

Algorithm 4 Loop decomposition of a given sound FW-net $WN = (P, T, F, i, o)$.

```

1:  $Connections \leftarrow \emptyset$ 
2:  $AcyclicNets \leftarrow \emptyset$ 
3: function DECOMPOSELOOPS( $WN = (P, T, F, i, o)$ )
4:   Identify  $Loops(WN)$ .
5:   if  $|Loops(WN)| = 0$  then
6:      $AcyclicNets \leftarrow AcyclicNets \cup \{WN\}$ 
7:     return  $AcyclicNets, Connections$ 
8:   for all  $L = (P_L, T_L, F_L) \in Loops(WN)$  do
9:     Identify loop entries  $Entries(L)$  and loop exits  $Exits(L)$ .
10:    Identify do-body  $DoBody(L)$  of  $L$ .
11:    Copy do-body and relink flows.
12:    Replace  $L$  with place  $p_L$  in  $WN$  and relink flows to  $p_L$ .
13:     $Connections \leftarrow Connections \cup \{(p_L, P_L \cup T_L)\}$ 
14:    Split  $L$  into loop fragments.
15:    Insert one source and one sink place to combine loop fragments.
16:    Create new net  $WN_L$  of it.
17:    DECOMPOSELOOPS( $WN_L$ )
18:  DECOMPOSELOOPS( $WN$ )
19:  return  $AcyclicNets, Connections$ 

```

in Figure 6 contains a loop with the nodes $\{b, B, c, d, C, e, D, f, E, Q, g, F\}$ at the top. This loop in turn contains a nested loop with the nodes $\{b, B, c, d, C, e, Q, g, F\}$. However, this nested loop is only revealed when the surrounding loop is decomposed. Loop decomposition handles such loops recursively.

The extracted loops with all their nodes and flows are decomposed by removing all incoming flows of loop exits. As a consequence, the loop disintegrates into at least one *loop fragment*. In the original method of loop decomposition [17], each fragment is extended to an own net. This is unnecessary in this context of sound FW-nets, since all fragments of the same loop are mutually exclusive [17], i. e., no node of one fragment can ever be in a concurrency relation with a node of another fragment of the same loop. Therefore, all fragments of a loop together get a new single source place and a new single sink place. For each loop entry and exit, a new transition is inserted connecting the source place with the entry/exit. For each loop exit, a new transition is inserted to connect it with the new sink place. In addition, all transitions previously in the preset of a loop exit are connected to the new sink place. The resulting “loop” net is sound and free-choice. In case that this *loop net* is still cyclic (in case of nested loops), loop decomposition can be recursively applied to this net again. This procedure finally terminates in only sound AFW-nets [17].

Figure 6 illustrates the decomposition of the cyclic FW-net on the left into its set of AFW-nets. The cyclic net contains two loops: One with place b as entry and f as exit in the top and one with place h as entry and j and k as exits. The net is decomposed into four acyclic nets (right). The net at the top is the loop-reduced version of the cyclic net. The do-bodies (loop 1: $\{b, B, c, d, C, e, D, Q, g, F\}$, loop 2: $\{h, H, i, I\}$) remained and the loops were replaced with loop places (α and β). Since the remaining do-body of loop 1 contains a nested loop with loop entries b and g and loop exit e , the recursive decomposition replaces this loop with loop place γ and its do-body $\{b, B, c, d, C\}$ remains. The acyclic α -loop net corresponds to the upper loop place α and spans from before its loop exit f to before f (D). This net contains also the nested loop γ and is decomposed accordingly. The β -loop net corresponds to the lower loop place β . It consists of two fragments (from exit j to before k and from exit k to before j) and is constructed by adding a source place, sink place, and the remaining transitions. Finally, the γ -loop net represents the nested loop spanning from e to before e .

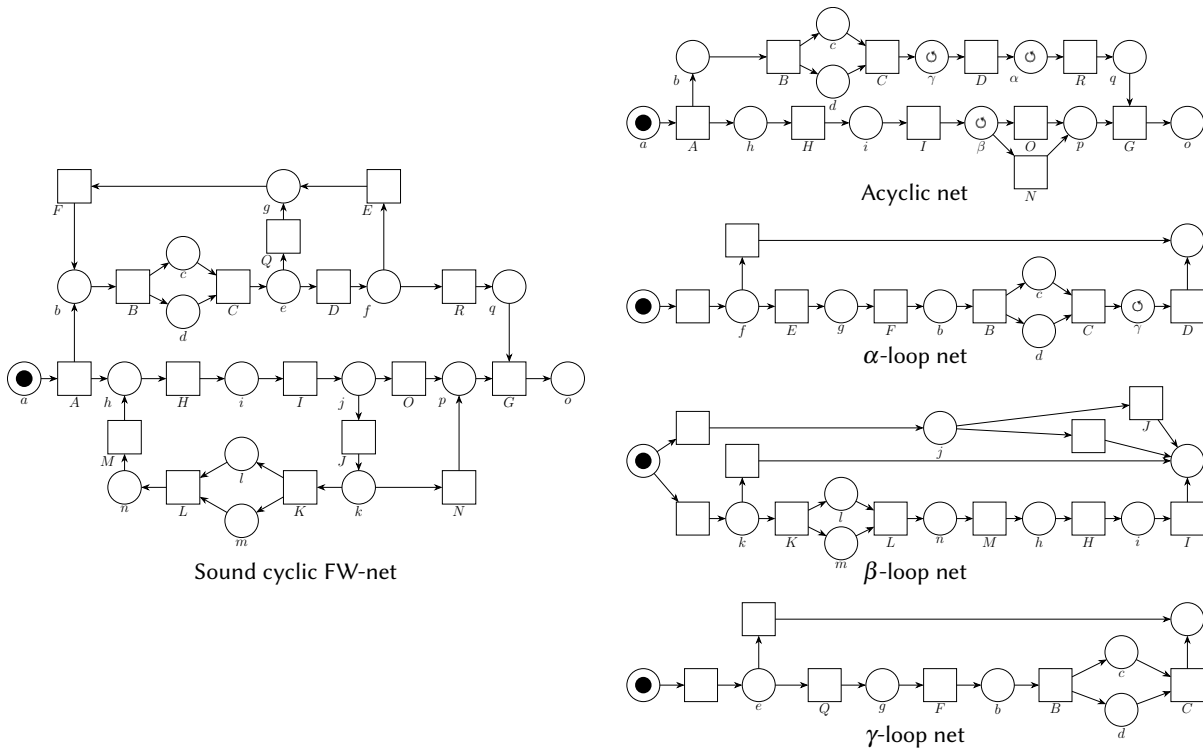


Figure 6: A sound cyclic FW-net with two loops and one nested loop (left), and its decomposition into four sound acyclic nets (right). The α - and β -loop nets represent both loops, which were reduced in the net above with (loop) places α and β . The γ -loop net represents the nested loop and has a corresponding loop place γ .

5.2. The Algorithm

The overall situation after loop decomposition is a set of sound AFW-nets, in which some *loop* places are linked to acyclic *loop* nets. It is important to understand for correctness that the replacement of (parts of) loops with loop places does not change the concurrency behavior. That is, in sound FW-nets, an entire loop (after copying the do-body) acts as a place “globally” [17]. For the computation of the concurrency relation it follows that each node in a concurrency relation with a *loop place* is concurrent with each node of the linked loop. Newly inserted transitions and places in the loop nets to facilitate the decomposition are *not* of interest, naturally. Nodes of loops in the do-body appear (at least) twice — once in the surrounding net and once in the corresponding loop net. For instance, places b and c in Figure 6 are in the AFW-net in the top and in the α -loop net. This does not have an influence on the final result, since if a node remaining in the surrounding net is in concurrency relation with another node, both nodes also are in concurrency relation in the original net. The same appears with nodes represented by the loop place being in concurrency relation with other nodes.

Algorithm 5 describes the *Concurrent Paths (CP)* algorithm. It computes the adjacency set R of concurrent nodes. This is initialized with an empty set in line 2. Line 3 calls Algorithm 4 to decompose net WN into a set of acyclic nets $AcyclicNets$ and a *Connections* relation. Lines 4–5 compute (and extend) R for each acyclic net WN_a with Algorithm 2. Lines 6–16 replace concurrency relations between loop places and other nodes. In doing this, it considers each connection between a loop place l and a linked loop net A (as combination of places and transitions) in line 6. For l , it considers each node c that is concurrent to l (line 7). If c is also a loop place with a linked loop net, the nodes of that loop net are assigned to B (lines 8–9). Otherwise, if c is an ordinary node, B only consists of $\{c\}$ (line 11). Furthermore, although c is concurrent to the loop place l , it is not of interest since l is “virtual” (line 12). Then, each node a in A is concurrent to each node in B (lines 13–14). In addition, each node b in B is concurrent to each node in A (lines 15–16).

The time complexity of the algorithm depends on the time complexity of loop decomposition as well as on the (possible) increase of the problem size. Loop decomposition is achievable in $O(|P|^2 + |P| \cdot |T| +$

Algorithm 5 The *Concurrent Paths (CP)* algorithm: A cyclic version of the algorithm to determine $\|$ for a sound FW-net $WN = (P, T, F, i, o)$.

```

1: function CONCURRENTPATHS( $WN = (P, T, F, i, o)$ )
2:    $R \leftarrow \emptyset$ 
3:    $AcyclicNets, Connections \leftarrow$  DECOMPOSELOOPS( $WN$ )
4:   for all  $WN_a \in AcyclicNets$  do
5:      $R \leftarrow$  DETERMINECONCURRENCY( $WN_a, R$ )
6:   for all  $(l, A) \in Connections$  do
7:     for all  $c \in R(l)$  do
8:       if  $c \in Connections$  then
9:          $B \leftarrow Connections(c) \cap (P \cup T)$ 
10:      else
11:         $B \leftarrow \{c\}$ 
12:         $R(c) \leftarrow R(c) \setminus \{l\}$ 
13:        for all  $a \in A$  do
14:           $R(a) \leftarrow R(a) \cup B$ 
15:        for all  $b \in B$  do
16:           $R(b) \leftarrow R(b) \cup A$ 
17:   return  $R$ 

```

$|P| \cdot |F|$) [17, 18]. In the worst case of many nested loops, the problem size after loop decomposition may increase quadratically, i. e., instead of checking one net of size $|P| + |T| + |F|$, $|P|$ nets of size $|P| + |T| + |F|$ must be investigated as a result of the decomposition (in general, the number of (nested) loops in a net is limited by a low constant). Thus, the complexity of checking a single cyclic net may increase to $O(|P|^2 + |T| \cdot |P| + |P| \cdot |F|)$ in the worst case. In this worst case, the application of Algorithm 2 on $|P|$ nets finally leads to a $O(|P|^3 + |P| \cdot |T|^2)$ and, therefore, cubic time complexity. For this reason, Algorithm 5 has the same time behavior as the *KovEs* algorithm in the worst case. Although it seems that Algorithm 5 has no benefit regarding the *KovEs* algorithm, there are several reasons why there are situations, in which the algorithm has its advantages:

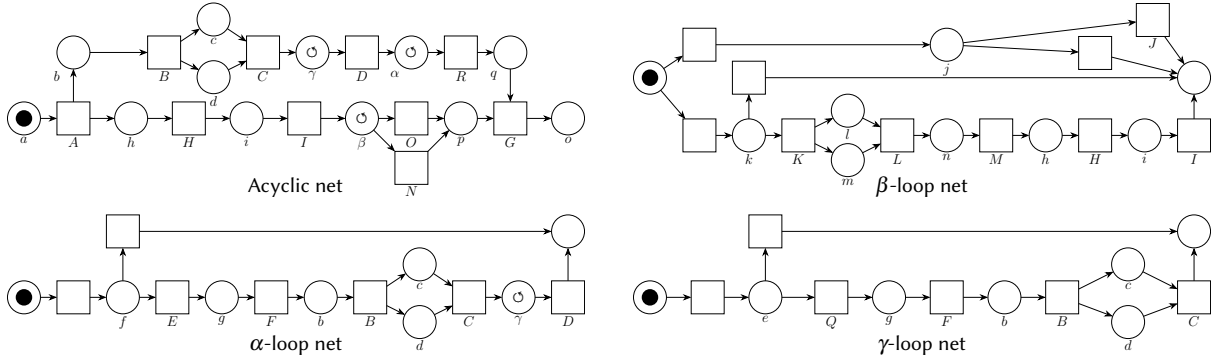
- (i) The algorithm can be parallelized. At first, each resulting acyclic net can be analyzed in parallel. The combination of the results is possible in at most quadratic time. At second, once *HasPath* is computed for a net in at most quadratic time, all transitions can be computed in parallel. The combination of the results is at most quadratic as well.
- (ii) It is relatively rare that nets have many (nested) loops [17]. In addition, if a loop has only one entry, it is not necessary to consider the do-body. The same holds true if the do-body does not contain any converging transition.
- (iii) If the net has a high degree of concurrency involving many nodes, the algorithm should have a performance benefit compared to *KovEs*.
- (iv) Of course, if the net is acyclic, the algorithm is faster.

5.3. Example

To illustrate how Algorithm 5 works, we use the cyclic example net of Figure 6. At first, loop decomposition decomposes the cyclic net into the acyclic nets on the right side of Figure 6. Subsequently, it performs for each of the acyclic nets the acyclic *CP* Algorithm 2. Table 1 contains the relation *HasPath* and the temporary relation R for each of the acyclic nets and most of their nodes. For example, if transition A is investigated, the pair (b, h) is considered. The sets $HasPath(b)$ and $HasPath(h)$ can be found in Table 1.

Table 1

The table shows for each acyclic net and their nodes being illustrated in the top, the relations *HasPath* and *R*. For *HasPath*, we omitted the mention for nodes, which are not in concurrency relation with another node. Both relations were derived after performing Algorithm 2.



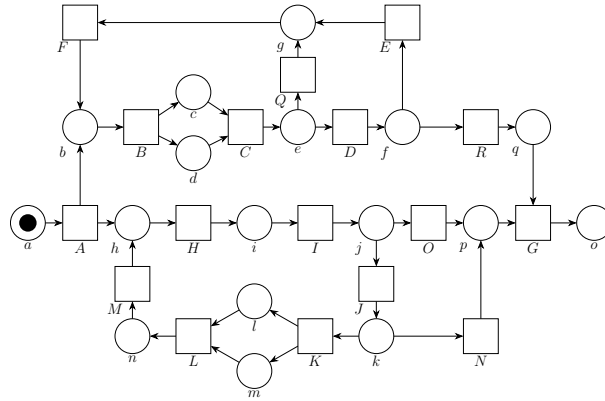
Net	Node(s)	<i>HasPath</i>	<i>R</i> ()
Acyclic	a, A, G, o	[Not of interest for this example]	\emptyset
	b	$b, B, c, d, C, \gamma, D, \alpha, R, q, G, o$	$h, H, i, I, \beta, O, N, p$
	B	$b, c, d, C, \gamma, D, \alpha, R, q, G, o$	$h, H, i, I, \beta, O, N, p$
	c	$c, C, \gamma, D, \alpha, R, q, G, o$	$d, h, H, i, I, \beta, O, N, p$
	d	$d, C, \gamma, D, \alpha, R, q, G, o$	$c, h, H, i, I, \beta, O, N, p$
	C	$C, \gamma, D, \alpha, R, q, G, o$	$h, H, i, I, \beta, O, N, p$
	γ	$\gamma, D, \alpha, R, q, G, o$	$h, H, i, I, \beta, O, N, p$
	D	D, α, R, q, G, o	$h, H, i, I, \beta, O, N, p$
	α	α, R, q, G, o	$h, H, i, I, \beta, O, N, p$
	R	R, q, G, o	$h, H, i, I, \beta, O, N, p$
	q	q, G, o	$h, H, i, I, \beta, O, N, p$
	h	$h, H, i, I, \beta, O, N, p, G, o$	$b, B, c, d, C, \gamma, D, \alpha, R, q$
	H	$H, i, I, \beta, O, N, p, G, o$	$b, B, c, d, C, \gamma, D, \alpha, R, q$
	i	$i, I, \beta, O, N, p, G, o$	$b, B, c, d, C, \gamma, D, \alpha, R, q$
	I	I, β, O, N, p, G, o	$b, B, c, d, C, \gamma, D, \alpha, R, q$
	β	β, O, N, p, G, o	$b, B, c, d, C, \gamma, D, \alpha, R, q$
O	O, p, G, o	$N, b, B, c, d, C, \gamma, D, \alpha, R, q$	
N	N, p, G, o	$O, b, B, c, d, C, \gamma, D, \alpha, R, q$	
p	p, G, o	$b, B, c, d, C, \gamma, D, \alpha, R, q$	
α	$f, E, g, F, b, B, C, \gamma, D$	[Not of interest for this example]	\emptyset
	c	c, C, γ, D	d
	d	d, C, γ, D	c
β	$j, J, k, K, L, n, M, h, H, i, I$	[Not of interest for this example]	\emptyset
	l	l, L, n, M, h, H, i, I	m
	m	m, L, n, M, h, H, i, I	l
γ	e, Q, g, F, b, B, C	[Not of interest for this example]	\emptyset
	c	c, C	d
	d	d, C	c

It is valid that $HasPath(b) \setminus HasPath(h) = \{b, B, c, d, C, \gamma, D, \alpha, R, q\}$. For each of these nodes, we can extend the relation to, e. g., $R(b) = R(b) \cup (HasPath(h) \setminus HasPath(b)) = R(b) \cup \{h, H, i, I, \beta, O, N, p\}$.

Once all acyclic nets are investigated, the results can be combined. For the upper acyclic net in Figure 6, there are three loop places α , β , and γ . Let us take α as an example: α is already concurrent to h, H, i, I, β, O, N , and p (cf. Table 1). Each of these nodes is considered by Algorithm 5, lines 7–16. It may start with node h , which is not a loop place but an ordinary node. Therefore, $B \leftarrow \{h\}$ and h is not concurrent to α anymore (lines 10–12). However, each node in the α -loop net (Figure 6), is concurrent to h (lines 13–14); and h (in B) is concurrent to each node of the α -loop net. The same holds true for the case of β : It is a loop place so that B contains all nodes of the β -loop net (Figure 6), lines 8–9. That means, each node of the α -loop net is concurrent to each node of the β -loop net and, naturally, vice versa (lines 13–16). Table 2 summarizes the R (||) relations for each node of the original net of Figure 6.

Table 2

The nodes of the net in the top being in a concurrency relation \parallel after performing Algorithm 5 and combining the R relations in Algorithm 2.



Node	$R(\parallel)$	Node	$R(\parallel)$
a	\emptyset	h	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
A	\emptyset	H	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
b	$h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	i	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
B	$h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	I	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
c	$d, h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	j	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
d	$c, h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	J	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
C	$h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	k	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
e	$h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	K	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
D	$h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	l	$m, b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
f	$h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	m	$l, b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
E	$h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	L	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
g	$h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	n	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
F	$h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	M	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
Q	$h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	N	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
R	$h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	O	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
q	$h, H, i, l, j, J, k, K, l, m, L, n, M, N, O, p$	p	$b, B, c, d, C, e, D, f, E, g, F, Q, R, q$
G	\emptyset	o	\emptyset

6. Evaluation

The *KovEs* algorithm and the presented *CP* algorithm in Section 5 have been implemented in a simple script-based algorithm (PHP) for the purpose of evaluation. The implementation is open-source and available on GitHub¹. The following experiments were conducted on a machine with an Intel® Core™ i7 CPU with 4 cores, 16 GB of main memory, and Microsoft Windows 11 Professional. PHP was used in version 8. We performed all measures for 10 times, removed the best and worst times, and used the mean values of all the remaining runs.

Both algorithms were applied to a well-known dataset, namely the *IBM Websphere Business Modeler* dataset [19], which consists of 1,368 files and is referred to as *IBM* hereafter. Only 644 nets of the IBM dataset can be investigated since the algorithm requires sound FW-nets. The nets were available as Petri Net Markup Language (PNML) models. The comparison of both algorithms was restricted to the identification of concurrent places instead of concurrent places and transitions, since the *KovEs* algorithm cannot find concurrent transitions without modifications. The nets under investigation are small (75% with less or equal 58 nodes) to big (with a maximum of 546 nodes). Places are more frequent than transitions (approx. $61\% \pm 4\%$ of all nodes are places). With regard to the number of nodes, a net has approx. $111\% \pm 11.5\%$ flows.

Both algorithms find the same places being concurrent for all suitable nets of the IBM dataset. Although the total number of nodes $|N|$ in a net bounds the number of concurrent nodes $|\parallel| \leq |(P + T)^2|$, the number of nodes ($R^2 = 0.2$), places ($R^2 = 0.42$), and transitions ($R^2 = 0.06$) do not well explain the

¹<https://github.com/guybrushPrince/cp>

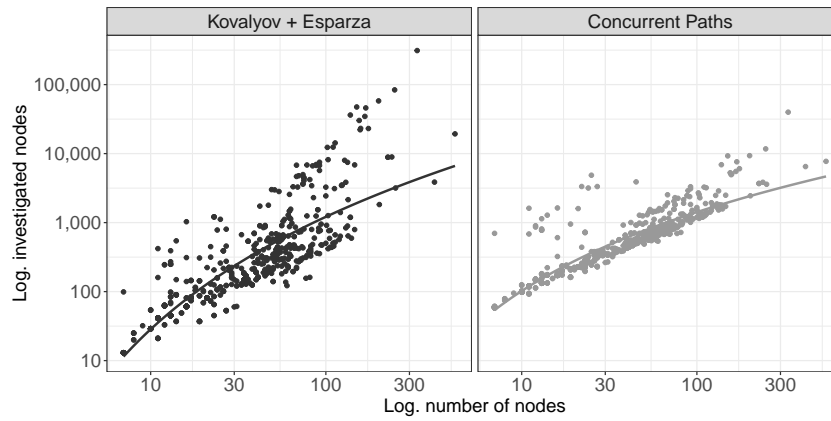


Figure 7: The number of investigated nodes during computation compared to the number of nodes in the nets.

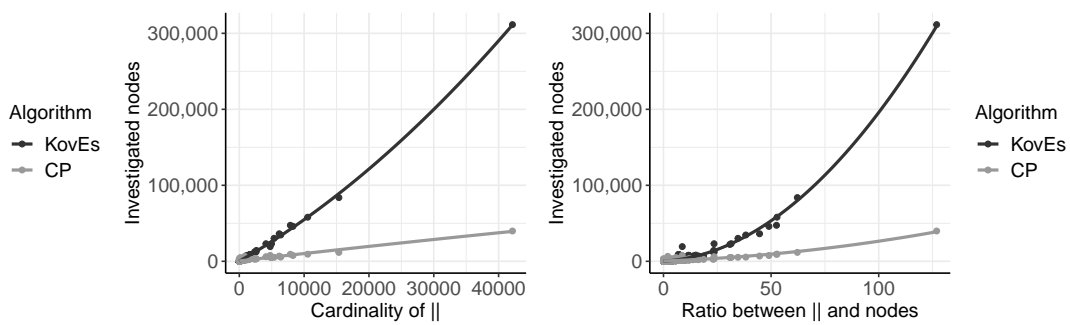


Figure 8: The number of investigated nodes during computation compared to the number of pairs in concurrency relation (left) and the ratio between the number of pairs and the number of nodes (right).

degree of concurrency in a net by applying a linear regression considering whether $|| \sim |(P + T)^2|$, $|| \sim |P^2|$, or $|| \sim |T^2|$, respectively.

The overall goal was to construct a more efficient algorithm than the *KovEs* algorithm. For this reason, we have compared the times both algorithms need to compute all concurrency relations for a net. The *CP* algorithm was overall faster for the entire dataset. It just needs 285 [ms] to compute 192,170 pairs of places being in relation. On contrary, the *KovEs* algorithm requires 14,100 [ms] for doing the same job, i. e., the *CP* algorithm is approx. 50 times faster. On closer inspection, *KovEs* has its benefits for nets without a high degree of concurrency. For these cases, it performs better than *CP*. Figure 7 shows a chart comparing the number of investigated nodes during both algorithms in relation to the number of nodes in the net. The y-axis is scaled logarithmically. The *CP* algorithm has, of course, a higher “start up” number of nodes to investigate, because it visits at least each node and flow once for the computation of paths. *KovEs* instead directly starts with the computation of the concurrency relations and, therefore, has no start up number of nodes and has a better performance for nets with less concurrent nodes. Figure 8 illustrates the number of investigated nodes in relation to the number of relations. The chart reveals that the computational load of *KovEs* seems to increase slightly quadratically if the number of nodes in relation increase. Instead, the load of *CP* seems to increase only linearly for a higher number of relations. This correlation between the computational load and the number of relations becomes more obvious if we compare the computation time in relation to the number of nodes being concurrent as it is done in Figure 9. The computation time of the *KovEs* algorithm seems to increase quadratically to the number of concurrent nodes; instead, the computation time of *CP* seems to increase just linearly.

Only 40 out of 644 nets of the IBM dataset are cyclic. The acyclic nets are investigated in 241 [ms] with *CP* and in 14,084 [ms] with *KovEs*. In fact, one net with approx. 42k concurrent pairs alone needs 10,500 [ms] for *KovEs* but just 45 [ms] with *CP*. *CP* is faster than *KovEs* for 87 nets. In sum, *CP* needs 171 [ms] and *KovEs* needs 14,06 [ms] for these 87 nets (speedup factor approx. 82), whereas for the

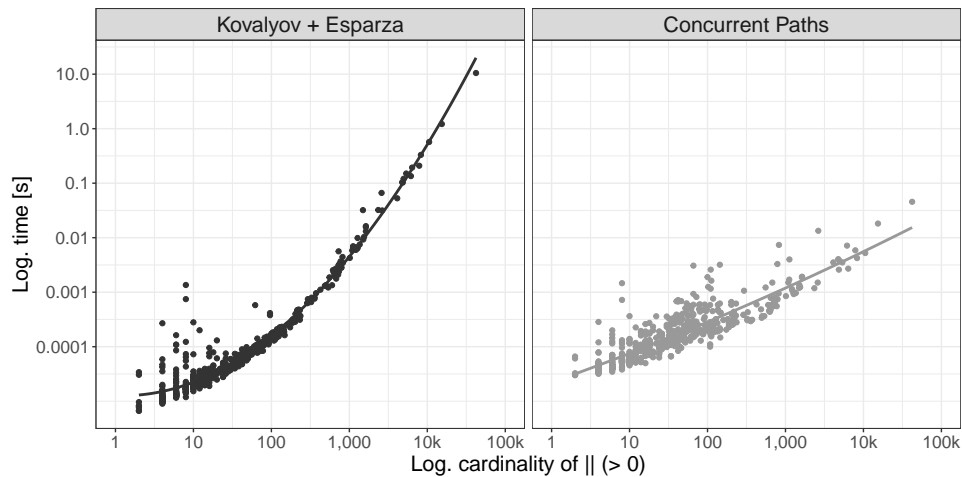


Figure 9: Computation time regarding the number of pairs of nodes in concurrency relation.

other 557 nets, *CP* needs 114 [ms] and *KovEs* needs 41 [ms] (slowdown factor approx. 3). The 87 nets comprise approx. 172k of pairs in relation against approx. 20k for the others. Therefore, the 87 nets have a much stronger computational intensity. *KovEs* is faster for the cyclic nets with 16 [ms] against 43 [ms] for *CP*. This is reasonable because of three reasons: (1) *CP* has a higher start up time especially for cyclic nets by performing a quadratic loop decomposition, (2) *CP* has the same worst-case cubic runtime complexity like *KovEs*, and (3) the ratio of concurrent nodes to the number of nodes is small for the 40 nets; there are just approx. 2.5 times of nodes in relation compared to the total number of nodes (i. e., a net with 100 nodes has around 250 pairs of nodes being concurrent). If this ratio is higher, *CP* benefits against *KovEs*. For showcasing this, the acyclic net with approx. 42k pairs of nodes in relation was surrounded with a simple loop. Although *CP* has to perform loop decomposition, the computation times are similar to those of the acyclic case — it shows that *CP* is more efficient than *KovEs* if a net contains many concurrent pairs of nodes.

7. Conclusion

Concurrency detection identifies pairs of nodes that may be executed in parallel. Knowing concurrent places and transitions in Petri nets is essential to understand their behavior and is crucial as the base for ongoing analysis. Since such nets are potentially large and complex with many pairs of concurrent nodes, efficient algorithms are necessary. This paper extends the palette of concurrency detection algorithms with the *Concurrent Paths* (*CP*) algorithm for sound and (simple) free-choice workflow nets. For acyclic nets, the algorithm performs in quadratic time $O(P^2 + T^2)$ with P the number of places and T the number of transitions of a net. The algorithm requires a cubic time complexity in the worst-case for cyclic nets, $O(P^3 + PT^2)$. Although this seems not to be an improvement of the algorithm of Kovalyov and Esparza (*KovEs*) (which needs a cubic time complexity for live, bounded, and extended free-choice nets and, therefore, a wider range of net classes), parallelizing *CP* is straight-forward and the worst-case of *CP* appears significantly less frequent than the worst-case of *KovEs* and can be assumed to have just a constant impact on computation time for most nets. An evaluation of *CP* on a benchmark of nets showed strong benefits on nets with a high degree of concurrency and only small disadvantages on nets with a low degree of concurrency.

This paper enables Petri net analysis to be more efficient, especially, in cases of a high degree in concurrency of nets. Although single nets could be analyzed efficiently with *KovEs*, performing concurrency detection on a large set of nets (e.g., for indexing in a database) may require much time. *CP* reduces the effort and enables more efficient strategies to compute other properties of nets such as causality, exclusivity, etc. As a side effect, related research areas such as business process management

and information systems research profit from the new technique as they utilize nets for analysis.

For future work, we plan to apply *CP* to efficiently derive all relations in the 4C spectrum [2] for sound free-choice workflow nets. This would allow for an efficient indexing of nets in querying languages for nets, which is especially beneficial in business process management.

References

- [1] R. M. Dijkman, M. L. Rosa, H. A. Reijers, Managing large collections of business process models - current techniques and challenges, *Comput. Ind.* 63 (2012) 91–97. doi:10.1016/J.COMPIND.2011.12.003.
- [2] A. Polyvyanyy, M. Weidlich, R. Conforti, M. L. Rosa, A. H. M. ter Hofstede, The 4C Spectrum of fundamental behavioral relations for concurrent systems, in: G. Ciardo, E. Kindler (Eds.), *Application and Theory of Petri Nets and Concurrency - 35th International Conference, PETRI NETS 2014, Tunis, Tunisia, June 23-27, 2014. Proceedings*, volume 8489 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 210–232. doi:10.1007/978-3-319-07734-5_12.
- [3] M. Weidlich, J. Mendling, M. Weske, Efficient consistency measurement based on behavioral profiles of process models, *IEEE Trans. Software Eng.* 37 (2011) 410–429. doi:10.1109/TSE.2010.96.
- [4] A. Kovalyov, Concurrency relations and the safety problem for Petri nets, in: K. Jensen (Ed.), *Application and Theory of Petri Nets 1992, 13th International Conference, Sheffield, UK, June 22-26, 1992, Proceedings*, volume 616 of *Lecture Notes in Computer Science*, Springer, 1992, pp. 299–309. doi:10.1007/3-540-55676-1_17.
- [5] A. Kovalyov, J. Esparza, A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs, Sonderforschungsbereich 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen TUM-19528, SFB-Bericht Nr. 342/15/95 A, Institut für Informatik, Technische Universität München, München, Germany, 1995.
- [6] M. Weidlich, A. Polyvyanyy, J. Mendling, M. Weske, Efficient computation of causal behavioural profiles using structural decomposition, in: J. Lilius, W. Penczek (Eds.), *Applications and Theory of Petri Nets, 31st International Conference, PETRI NETS 2010, Braga, Portugal, June 21-25, 2010. Proceedings*, volume 6128 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 63–83. doi:10.1007/978-3-642-13675-7_6.
- [7] N. L. Ha, T. M. Prinz, Partitioning behavioral retrieval: An efficient computational approach with transitive rules, *IEEE Access* 9 (2021) 112043–112056. doi:10.1109/ACCESS.2021.3102634.
- [8] I. Weber, J. Hoffmann, J. Mendling, Beyond soundness: on the verification of semantic business process models, *Distributed Parallel Databases* 27 (2010) 271–343. doi:10.1007/S10619-010-7060-9.
- [9] J. Esparza, S. Römer, W. Vogler, An improvement of McMillan’s unfolding algorithm, *Formal Methods Syst. Des.* 20 (2002) 285–310. doi:10.1023/A:1014746130920.
- [10] A. Polyvyanyy, Structuring process models, Ph.D. thesis, University of Potsdam, 2012. URL: <http://opus.kobv.de/ubp/volltexte/2012/5902/>.
- [11] C. Favre, D. Fahland, H. Völzer, The relationship between workflow graphs and free-choice workflow nets, *Inf. Syst.* 47 (2015) 197–219. doi:10.1016/j.is.2013.12.004.
- [12] T. M. Prinz, J. Klaus, N. R. van Beest, Pushing the limits: Concurrency detection in acyclic, live, and 1-safe free-choice nets in $O((P + T)^2)$, *CoRR abs/2401.16097* (2024). doi:10.48550/ARXIV.2401.16097. arXiv:2401.16097, Technical Report.
- [13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009.
- [14] J. Desel, J. Esparza, *Free Choice Petri Nets*, pbk version ed. edition ed., Cambridge University Press, 2008.
- [15] W. M. P. van der Aalst, Verification of workflow nets, in: P. Azéma, G. Balbo (Eds.), *Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN ’97, Toulouse, France, June*

- 23-27, 1997, Proceedings, volume 1248 of *Lecture Notes in Computer Science*, Springer, 1997, pp. 407–426. doi:10.1007/3-540-63139-9_48.
- [16] H. M. W. Verbeek, T. Basten, W. M. P. van der Aalst, Diagnosing workflow processes using woflan, *Comput. J.* 44 (2001) 246–279. doi:10.1093/comjnl/44.4.246.
- [17] T. M. Prinz, Y. Choi, N. L. Ha, Understanding and decomposing control-flow loops in business process models, in: C. D. Ciccio, R. M. Dijkman, A. del-Río-Ortega, S. Rinderle-Ma (Eds.), *Business Process Management - 20th International Conference, BPM 2022, Münster, Germany, September 11-16, 2022, Proceedings*, volume 13420 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 307–323. doi:10.1007/978-3-031-16103-2_21.
- [18] T. M. Prinz, Y. Choi, N. L. Ha, Soundness unknotted: Efficient algorithm for process models with inclusive gateways by loosening loops, *SSRN* (2023). doi:10.2139/ssrn.4504060, preprint.
- [19] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, K. Wolf, Analysis on demand: Instantaneous soundness checking of industrial business process models, *Data Knowl. Eng.* 70 (2011) 448–466. URL: https://web.archive.org/web/20131208132841/http://service-technology.org/publications/fahlandfjklvw_2009_bpm. doi:10.1016/j.datak.2011.01.004, last visited in Jun. 2024.