Inferring Activity Concurrency Relations from Incomplete Event Logs

César Barrón-Rubio[1], Ernesto López-Mellado[1,*]

[1]CINVESTAV Unidad Guadalajara. Av. Del Bosque 1145. Col El Bajío, 45019 Zapopan Jal. México

Abstract

In process discovery methods that build workflow nets (WF-nets), computing activity concurrency is essential to achieve the construction of a suitable model. The condition $a||b \leftarrow ab + ba$, used to determine the concurrency relation between two activities is surpassed when the event log is incomplete. This paper presents a technique for deducing additional concurrency relations between activities from incomplete event logs. The proposed technique is based on the detection of repetitive patterns within the traces in the event log; using these patterns, new logs in which the traces do not have repeated activities are derived. Afterward, a set of partial order structures is built from these event logs and then concurrency relations are straightforwardly obtained. Finally, we use a heuristic to determine concurrent relations between activities belonging to different repetitive patterns. The technique performs as a concurrency oracle; it has been implemented and tested on artificial event logs generated by WF-nets with diverse structures. Experiments show that the proposed oracle extracts more concurrent relations than other methods.

Keywords

Incomplete Event Log, Concurrency Oracle, Process Mining

1. Introduction

Extracting concurrency relationships from event logs is an increasingly relevant problem in Process Mining, mainly when not all direct follows relationships have been recorded; that is when the event log is *incomplete*. A function that takes an event log as input and returns a set of pairs of concurrent activities is called *concurrency oracle* [1]. This paper proposes a concurrency oracle for dealing with incomplete event logs.

Usually, the input to Process Discovery methods is an event log, where concurrency becomes explicit only until a process model (usually a Petri net) is discovered. However, there is a recent approach known as *partial-order-based Process Mining* [2], which considers input logs as *partial languages* (labeled partial orders). This approach has been mostly exploited in discovery methods based on the theory of regions [3, 4, 5]. Regions of partial languages have been investigated for two decades, obtaining important results that increasingly narrow the difference in execution times between region-based methods and other discovery approaches.

However, techniques based on partial orders are not restricted to discovery; tasks such as *conformance checking* and *process enhancement* have also been addressed. Even in [6], Dumas and

Algorithms & Theories for the Analysis of Event Data and Petri Nets for Twin Transition 2023

*Corresponding author.

cesar.barron@cinvestav.mx (C. Barrón-Rubio); e.lopez@cinvestav.mx (E. López-Mellado)

García-Bañuelos proposed reloading all Process Mining operations as operations between Prime Event Structures, labeled partial orders, assembled through their common prefixes, preserving conflicts. In [2], a summary of the works that start from a partial language is presented.

This trend has generated the need to extract concurrency relations directly from the event log since the usual way of transforming an event log into a set of partial orders (see [5], for example) requires knowing the concurrency relations beforehand. Despite the relevance of this topic, little research has been done on it.

The most used oracle, called α-oracle, is derived from the concurrency relationship of the α-algorithm [7]. In this, a pair of activities, x and y, are considered concurrent if both subtraces xy and yx are present in the event log. Concurrent relations used in the extensions of the α-algorithm [8, 9] are used too as oracles. In [10], the authors proposed using a domain expert as an oracle; a human expert points concurrency between activities. This approach can be used to refine another oracle, like the α.

If the event log contains information about the life cycle of activities, the life-cycle oracle can be implemented [5]. A pair of activities are concurrent if their life cycles overlap. In [11], one of the early works on process discovery, four statistical metrics are used to determine concurrency between activities; therefore, it can be considered an oracle.

The effectiveness of a concurrency oracle strongly depends on the *completeness* of the event log. Unfortunately, real-life event logs may be non-complete, as was pointed out in the challengers of the Process Mining Manifesto [12].

Only a few concurrency relationships can be extracted from an incomplete event log. This situation creates some problems. For example, the discovered models will lack *generalizability*; that is, they will not be able to generate unrecorded behavior or will generate very little. That is especially true for region-based methods because they assume that only the recorded behavior should be in the process model.

Furthermore, discovery methods based on region theory ultimately consist of solving an Integer Linear Programming problem, where the size of the problem is a function of the size of the partial order relation. When transforming a set of traces (linearly ordered sets) into a set of partial orders, the size of the relation is reduced since the partial order relation does not include relations between concurrent activities, while these are included in the traces. Therefore, the fewer concurrency relations we know, the more elements the partial order relations will have. The latter translates into a longer execution time.

This paper presents a novel Concurrency Oracle that deals with incomplete event logs. This method obtains a set of pairs of concurrent activities, including some that were not explicitly recorded. Our proposal is based on detecting cyclic components of the process and projecting the event log on their activities, obtaining new ”event logs” in which all the activities present in these logs are in the same cycle. These new event logs are transformed in a specific way so that activities do not appear more than once in each trace. Once we have such a set of traces without repeated activities, we can find the concurrency relationships using an algorithm from the early days of Process Mining. Experimental tests using artificial event logs showed a good performance in finding unrecorded concurrent relations. However, in some cases, fake concurrences are determined.

2. Preliminaries

This section contains the notions and notation used in the proposal. Concurrency oracles are independent of the process modeling notation. However, we use Petri nets for the running examples, as is customary in the Process Mining literature. Therefore, we began this section by presenting basic definitions of Petri nets. Later, formal definitions of concepts related to Process Mining are presented. Moreover, we recall an algorithm to build a directed acyclic graph from an event log.

2.1. Petri nets, and workflow nets

Definition 2.1 An ordinary *Petri net structure* is a triple $G = (P, T, F)$ where P is a finite set of places, T is a finite set of transitions such that $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (P \times T)$ is a set of direct arcs, called the flow relation. A *marked Petri net* is a pair (N, M_0), where $N = (P, T, F)$ is a Petri net structure and $M_0 \in B(P)$ is a multi-set over P denoting the marking of the net. The elements of multisets will be written between square brackets.

Places in a Petri net structure are drawn as circles, while transitions are drawn as rectangles. The marking of the net is represented by *tokens*, black dots inside places. Figure 1 shows a marked Petri net.

Given a Petri net structure $N = (P, T, F)$ and $x \in T \cup P$, we call *preset* of x to the set $\bullet x = \{y \,|\, (y, x) \in F\}$. Respectively, the *postset* of x is $x\bullet = \{y \,|\, (x, y) \in F\}$. In a marked Petri net (N, M), the transition $t \in T$ can be *fired*, iff $\bullet t \subseteq M$. The red firing of t leads to a new marking $M^{'} = (M \setminus \bullet\, t) \uplus t\bullet$. Where $M \setminus \bullet\, t$ is the multiset formed by deleting of M the elements in $\bullet t$, and $\uplus$ is the sum of multisets. $M \overset{t}{\rightarrow} M^{'}$, denote that being in marking M, firing transition t led to marking $M^{'}$. The set of all reachable markings of (N, M) is denoted by $[N, M\rangle$.

Definition 2.2 Given a Petri net structure $N = (P, T, F)$. The *incidence matrix* of N is the matrix $C \in \{0, +1, -1\}^{|P| \times |T|}$ defined by $C_{ij} = -1$ if $p_i \in \bullet t_j \wedge p_i \notin t_j\bullet$, $C_{ij} = 1$ if $p_i \in t_j \bullet \wedge p_i \notin \bullet t_j$, and $C_{ij} = 0$ otherwise.

Definition 2.3 A *T-invariant* Y_i of a Petri net structure $N = (P, T, F)$ with incidence matrix C, is an integer solution to the equation $CY_i = \overline{0}$ such that $Y_i \geq \overline{0}$ and $Y_i \neq \overline{0}$. The support of Y_i, denoted as $\langle Y_i \rangle$ is the set of transitions whose corresponding entries in Y_i are strictly positive. A *T-component* $G(Y_i) = (P_i, T_i, F_i)$, is a subnet of N where $P_i = \bullet\langle Y_i \rangle \cup \langle Y_i \rangle\bullet$, $T_i = \langle Y_i \rangle$, $F_i = ((P_i \times T_i) \cup (T_i \times P_i)) \cap F$.

Proposition 1 Fundamental property of T-invariants

Let σ be a finite sequence of transitions of a net N which is enabled at a marking M. Then the Parikh vector $\overrightarrow{\sigma}$ is a T-invariant iff $M \overset{\sigma}{\rightarrow} M$. [13]

The Proposition 1 results are essential for our goal. This states that after executing all the transitions in a T-invariant (including repetitions), we arrive at the same marking from which we started. Thus, a T-component is a *possible* cycle of the net. In Figure 1, the subnet in red is the T-component induced by the T-invariant $[0\ \ 1\ \ 1\ \ 1\ \ 0\ \ 0]$.

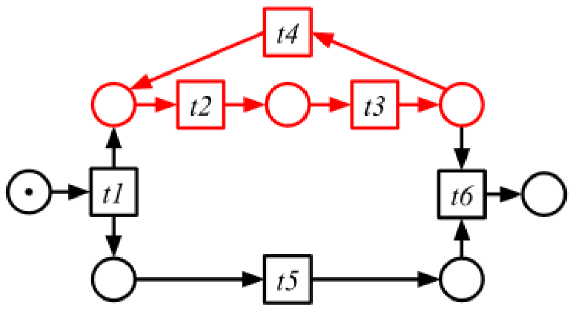

Figure 1: Graphical representation of a Petri net

Definition 2.4 A *labelled Petri net* is a tuple (P, T, F, A, l) where (P, T, F) is a Petri net structure, A is a set of activity labels, and $l : T \to A$, a labelling function that assigns activity names to transitions.

In this context, the labels will be the names of the process activities. The rest of the article assumes that no two transitions can have the same label. So, when we say label, transition, or activity, we mean the same thing.

Definition 2.5 Let $N = (P, T, F, A, l)$ be a labeled Petri net and $\bar{t}$ a transition not in $P \cup T$. N is a *workflow net* (WF-net) [14] if and only if:

1. P contains an input place i (source place) such that $\bullet i = \emptyset$,
2. P contains an output place o (sink place) such that $o\bullet = \emptyset$,
3. $\overline{N} = (P, T \cup \{\bar{t}\}, F \cup \{(o, \bar{t}), (\bar{t}, i)\}, A \cup \tau, l \cup \{(\bar{t}, \tau)\})$ is strongly connected.

In Process Mining, it is common to assume that WF nets are "well-formed". Specifically, it requires that a WF-net be *sound.*
Definition 2.6 Let $N = (P, T, F, A, l)$ be a WF-net with an input place i and an output place o. N is *sound* [14] if and only if:

1. $(N, [i])$ is safe, i.e., places cannot hold multiple tokens at the same time;
2. for any marking $M \in [N, [i]\rangle$, $o \in M$ implies $M = [o]$;
3. for any marking $M \in [N, [i]\rangle$, $[o] \in [N, M\rangle$;
4. $(N, [i])$ contains no dead transitions.

2.2. Event logs, and concurrency oracles

Events stored in event logs have several attributes, for example, case identifier, name of the activity executed, resources used, information about who performs the activity, start and end date, etc. For our purposes, it is enough to consider that the events have the name of the executed activity and a case identifier. Also, we assume that the events with the same case identifier are in a total order relationship. Thus, we use the following simplified definition of an event log.

Definition 2.7 Let A be a set of activity names. A trace σ is a finite sequence of elements of A. An *event log* is a set of traces.

An event log may contain noise or behavior that does not belong to the underlying process. However, in this work, we assume that the event logs are noise-free; that is, we consider that the process generated all the recorded behavior. In real-life event logs, this can be a huge assumption. However, there are preprocessing methods that mitigate this problem, for example, [15, 16]

Two essential concepts mentioned briefly in the introduction are the completeness of an event log and the concurrency between activities. Both are based on the direct-follows relation defined in [7].

Definition 2.8 Let L be an event log. a is *directly followed* by b, denoted by $a >_L b$, if and only if there is a trace $\sigma = \langle t_1, t_2, ..., t_n \rangle$ and $i \in \{1, ..., n-1\}$ such that $\sigma \in L$ and $t_i = a$ and $t_{i+1} = b$. The set $>_L = \{(x, y) \mid x, y \in A,\ x >_L y\}$ is the *direct follow relation.*

For a Petri net, we can also consider the direct follow relation between transitions.

Definition 2.9 Let t_1 and t_2 transitions of the marked Petri net (N, M_0) labeled by a, and b respectively. We say that a is *directly followed* by b if and only if there are reachable markings M, $M^{'}$ and $M^{''}$ of N such that $M \overset{t_1}{\to} M^{'} \overset{t_2}{\to} M^{''}$.

Note that, unlike the *causal relationship*, there is no need for a place connecting the transitions in this case.

Now, the usual definition of completeness is recalled from [7].

Definition 2.10 Let L be an event log extracted from the WF-net N. It is said that L is *complete* if the direct follows relation derived from L, coincides with that derived from N.

The completeness property is weaker than requiring that all possible traces be in the event log, which is impossible if the Petri net has cycles. However, even this weak definition rarely holds true in real-life settings. Therefore, it would be more meaningful to talk about the rate of completeness rather than completeness.

Definition 2.11 Let L be an event log extracted from the a WF-net N, with at least two transitions. By $\#(L, \to)$, we denote the number of pairs of activities in direct follow relation concerning L. Similarly, $\#(N, \to)$ denotes the number of pairs of activities in direct follow relation concerning N. The quotient $\frac{\#(L,\to)}{\#(N,\to)}$ is the *rate of completeness* of the log L with respect to the net N.

To the best of our knowledge, the above definition had not been explicitly stated in the literature. However, it was used in [17] to evaluate the rediscovery capability of the algorithm proposed there. The rate of completeness can measure the completeness of an event log. Note that the rate of completeness is 1 only if the log is complete in the usual sense mentioned earlier.

Intuitively, two activities are concurrent if there is neither a causal relationship nor a conflict between them. In a Petri net, we considered that two transitions are concurrent if:

Definition 2.12 A pair of transitions labeled by $a, b \in A$ are *concurrent* in the sound WF-net $(N, [i])$ if $\bullet a \cap \bullet b = \emptyset$, and if there is a marking $M \in [N, [i]\rangle$ such that transitions labeled by a and b can be fired, and after firing one of them, the other one can still be fired.

In agreement with [7], concurrency can be inferred from the log based on the direct-follow relation.

Definition 2.13 Let L be an event log and A the set of activities. If $a, b \in A$, we said that a is *concurrent* with b if $a >_L b$, and $b >_L a$. Concurrency is denoted by $a||_\alpha b$.

A formal definition of a concurrency oracle is given in [18]. However, it is enough for us to say that a concurrency oracle is "a black-box Boolean function that asserts whether a given pair of events are concurrent or not" [1].

Let O be a concurrency oracle. Abusing the notation, we will treat O as a family of two-element sets, such that $\{a, b\} \in O \Leftrightarrow a||b$. That is, we consider an oracle O as the set of pairs of concurrent activities.

2.3. Building directed acyclic graphs

In this subsection, we recall one of the earlier works of Process Discovery due to Agrawal et, al. [19]. The goal of that work was not to build a Petri net but a labeled directed graph. In particular, we summarize the second algorithm shown there, which assumes that the traces have no repeated activities. This algorithm is fundamental to our proposal; for this reason, it is described in detail through an example.

Example 1: We illustrate the Agrawal's algorithm using the log $L_1 = \{\langle a, b, c, f\rangle, \langle a, c, d, f\rangle, \langle a, d, e, f\rangle, \langle a, e, c, f\rangle\}$. (Example 7 of [19])

Firstly, a graph is created with vertices equal to the set of activities ($\{a, b, c, d, e, f\}$) and initially an empty set of arcs. Then:

1. For each trace of L_1, the transitive closure of the direct follow relation is added to the set of arcs. The resulting graph is shown in the Figure 2 on the left.
2. Arcs that appear in both directions are removed. In addition, arcs belonging to some strongly connected component of the graph are eliminated too. The resulting graph from step 2 is shown in the Figure 2 in the center.
3. For each trace, the subgraph induced by it is taken, and its transitive reduction is calculated. Arcs that do not belong to any of these transitive reductions are removed. The resulting graph from step 3 is shown in the Figure 2 on the right.

Figure 3 shows the transitive reductions of the step 3. These graphs are indeed the skeletons of *posets* (partial-ordered sets). *Posets* express Petri net executions as an alternative to traces, although the verification problem (checking if a *poset* is executable by a given Petri net) is more

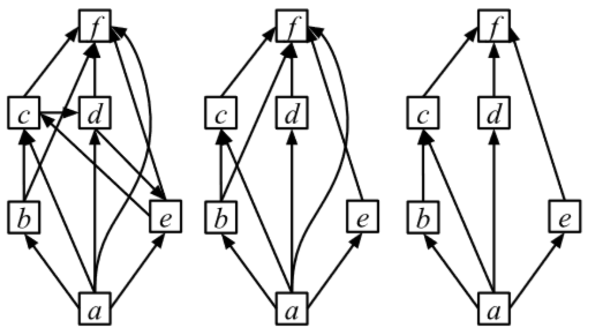

Figure 2: Steps of the Agrawal algorithm on L_1

complex [20]. In this context, the fact that two vertices are not related is interpreted as the activities involved (their labels) being concurrent. In the Example, (c, d), (d, e), and (c, e) are concurrent relationships. Thus, Agrawal's algorithm is useful to extract concurrency relations between activities from traces.

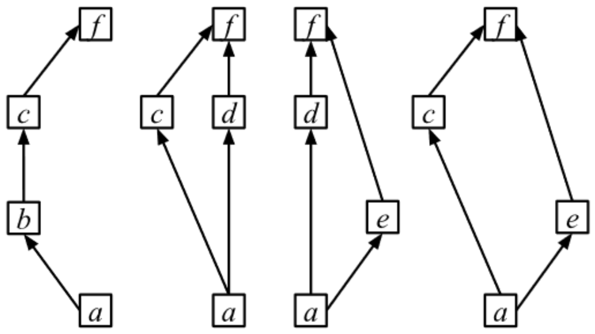

Figure 3: Graphical representation of the transitive reduction induced by traces of log L_1

3. Extracting Concurrency from Incomplete Event Logs

3.1. Searching concurrency

We hypothesize that concurrent relationships between activities within a cycle are not observed when analyzing the event log due to the interleaved occurrence of activities outside the cycle. However, if the part of the traces corresponding to a cycle is isolated, ignoring everything outside the cycle, the previously hidden concurrent relationships become observable.

Example 2: Consider the WF-net shown in Figure 4 in which transitions belonging to cyclic components are colored with different colors. From this net the following event log is extracted: $L_2 = \{\langle q, s, w, f, e, r, i, a, g, j, y, f, g, j, f, u, d, o, g, j, e, r, t, f, i, a, g, j, u, f, g, d, h, k, z, x, c\rangle, \langle q, w, s, f, e, r, y,$

$i, u, a, d, o, e, g, i, h, r, a, y, u, d, o, e, i, r, y, a, u, d, l, n\rangle, \langle q, s, w, e, i, r, a, y, u, d, o, f, e, g, i, h, a, r, d, y, u, o, e, i, a, r, y, u, d, o, e, i, a, r, t, u, d, l, n\rangle\}$.

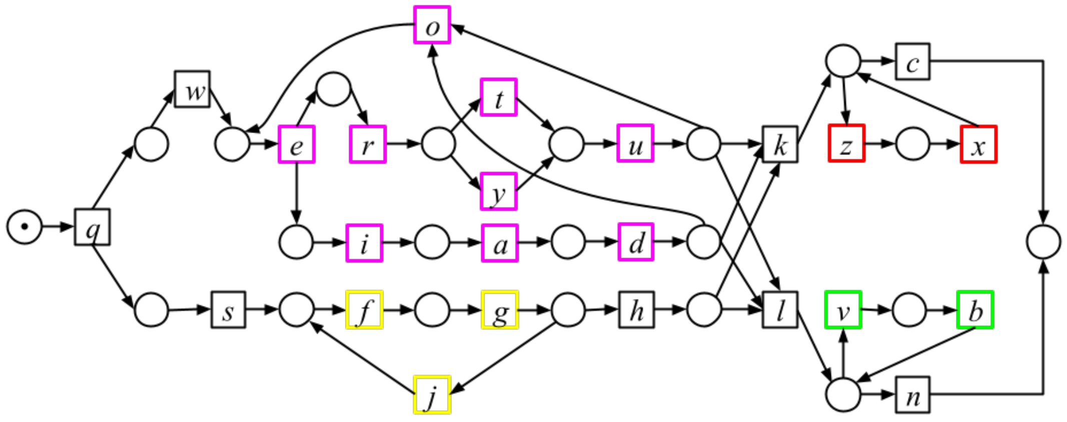

Figure 4: Petri net with the transitions belonging to the same cycle colored in the same color

If instead of considering the complete event log, we consider only the projection of the traces on one of the cycles, for example, the pink one, we obtain the traces $L_2' = \{\langle e, r, i, a, y, u, d, o, e, r, t, i, a, u, d\rangle, \langle e, r, y, i, u, a, d, o, e, i, r, a, y, u, d, o, e, i, r, y, a, u, d\rangle, \langle e, i, r, a, y, u, d, o, e, i, a, r, d, y, u, o, e, i, a, r, y, u, d, o, e, i, a, r, t, u, d\rangle\}$. From L_2' we can build an event log with no repeated activities in the traces (splitting the trace just before starting the cycle again). In our example, we get the log $L_2'' = \{\langle e, r, i, a, y, u, d, o\rangle, \langle e, r, t, i, a, u, d\rangle, \langle e, r, y, i, u, a, d, o\rangle, \langle e, i, r, a, y, u, d, o\rangle, \langle e, i, r, y, a, u, d\rangle, \langle e, i, r, a, y, u, d, o\rangle, \langle e, i, a, r, d, y, u, o\rangle, \langle e, i, a, r, y, u, d, o\rangle, \langle e, i, a, r, t, u, d\rangle\}$.

Notice that, L_2'' fulfill the assumptions of the Agrawal's algorithm. Hence, we obtain that (r, i), (i, y), (a, y), (a, u), (d, u), (y, d), (t, i), and (a, t) are concurrent relationships. Only the red ones were explicitly recorded.

3.2. A method for inferring hidden concurrences

3.2.1. Support concepts and operators

First, some concepts and operators on the event log needed for concurrency extraction are introduced. The concept of *repetitive dependency* introduced in [21] is also recalled.

Definition 3.1 Let L be an event log, and $\sigma \in L$ be a trace.

1. The *non-repetitive division* of σ, denoted as $d(\sigma)$, is a set of sub-traces $\{\sigma_1, \sigma_2, ..., \sigma_n\}$ such that:
   - $\sigma = \sigma_1\sigma_2 \cdots \sigma_n$,
   - σ_1 is the larger prefix of σ with no repeated activities, and
   - for $i \in \{2, ..., n\}$, σ_i is the larger prefix of τ without repetition of activities, where $\sigma = \sigma_1 \cdots \sigma_{i-1}\tau$.
2. The *non-repetitive event log* derived from L is $\mathscr{D}(L) = \bigcup_i d(\sigma_i)$; the union of all non-repetitive divisions of traces in L.

Notice that the non-repetitive division of a trace σ is unique; consequently, $\mathscr{D}(L)$ is unique for a given L and contains traces that have no repeated activities.

Example 3: Consider the event log $L_3 = \{\sigma_1 = \langle e, r, i, a, y, u, d, o, e, r, t, i, a, u, d\rangle,\ \sigma_2 = \langle e, r, y, i, u, a, d, o, e, i, r, a, y, u, d, o, e, i, r, y, a, u, d\rangle,\ \sigma_3 = \langle e, i, r, a, y, u, d, o, e, i, a, r, d, y, u, o, e, i, a, r, y, u, d, o, e, i, a, r, t, u, d\rangle\}$ extracted from WF-net in Figure 4. $d(\sigma_1) = \{\langle e, r, i, a, y, u, d, o\rangle,\ \langle e, r, t, i, a, u, d\rangle\}$. $\mathscr{D}(L_3) = \{\langle e, r, i, a, y, u, d, o\rangle,\ \langle e, r, t, i, a, u, d\rangle,\ \langle e, r, y, i, u, a, d, o\rangle,\ \langle e, i, r, a, y, u, d, o\rangle,\ \langle e, i, r, y, a, u, d\rangle,\ \langle e, i, r, a, y, u, d, o\rangle,\ \langle e, i, a, r, d, y, u, o\rangle,\ \langle e, i, a, r, y, u, d, o\rangle,\ \langle e, i, a, r, t, u, d\rangle\}$

The next definition was stated in [21], and it is the base for computing the T-invariants of a Petri net from their language.

Definition 3.2 Let L be an event log, with activity names in A. An activity x is repetitively dependent on y, denoted as $x \prec y$ iff y is always observed between two apparitions of x in $\sigma \in L$. If x has been observed at least twice in $\sigma \in L$, then $x \prec x$. The set of transitions from which x is repetitively dependent is given by the function $Rd(x) : T \to 2^T$; then $Rd(x) = \{y | x \prec y\}$. If x was observed at most once in each $\sigma \in L$, then $Rd(x) = \emptyset$.

The sets $Rd(x)$ are not T-invariant; however, it was shown in [21] that these are included in the support of at least one T-invariant, thus representing repetitive parts of the underlying Petri net.

Now, using the introduced concepts, the concurrency oracle can be stated.

3.2.2. General Approach

The algorithm is divided into two steps. In the first step, the goal is to build a tree. The idea is that the tree nodes represent the repeating parts of a WF-net. The root node represents the complete WF-net. A WF-net is not a repeating component, but if we add an artificial transition like point 3 of definition 2.5, we can consider it so. Its child nodes are the cycles contained in it that are not contained in other cycles. In turn, the children of these will be their inner cycles until reaching the cycles that do not include any cycle inside, which will be the tree leaves. The non-repeating part of the WF-net also forms a node. For the WF-net in Figure 4, the generated tree is shown in Figure 5.

Of course, we do not have a WF-net in advance. Figure 5 only illustrates the ideal case where we know the cycles. In real cases where we only have one event log, the tree we will build will have traces on the nodes instead of the subnets in Figure 5. To construct the traces corresponding to one cycle, we use the sets $Rd(t)$ from Definition 3.3.

The second step is to find the concurrency relationships one node at a time. The strategy is to take an in-depth tour. Once a tree leaf is reached, we break the cycles so that the traces generated by them do not have repeated activities. This allows us to find concurrency relationships using Agrawal's algorithm.

We repeat the same strategy as in the leaves as we go up through the nodes. The difference is that Agrawal's algorithm can now find concurrences between activities that belong to the child nodes. We resort to a heuristic to decide whether these concurrences are conserved.

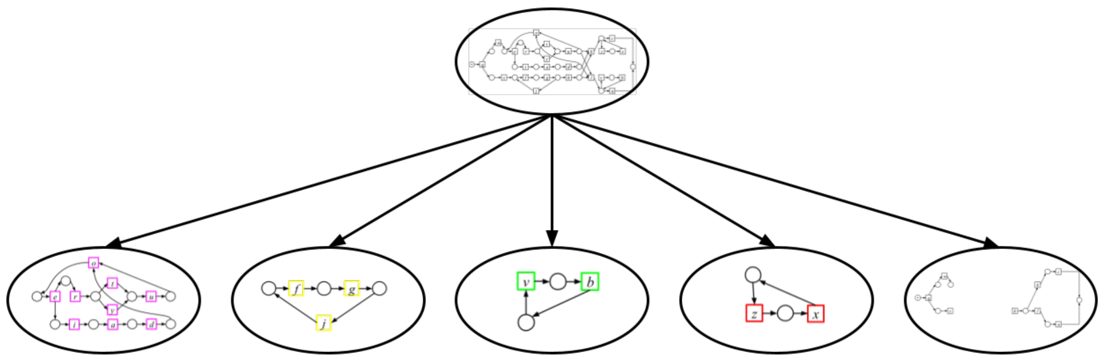

Figure 5: Decomposition of a Petri net in its cycles.

3.2.3. Step 1: Building the Tree

In sections 3.2.2 and 3.2.3, we described the method using the event log L_2 shown above as a running example.

This section shows the first step of our algorithm: the decomposition of a process into its repetitive components and the arrangement of these in the form of a tree. Unlike Figure 5, we do not have a WF-net to extract the cycles from there; all we have is an incomplete event log. We will try with this log to approximate the tree we would obtain if we had a model.

The nodes will contain sets of traces, and we define the root node as the one that contains the original event log, L_2.

To find its child nodes, we proceed as follows: we find the first repeated activity in any of the traces; in our example, this activity is f. In this case, f is the first activity repeated in the first trace, although we could consider any other trace.

This activity f represents the "start" of the cycle. We compute the set of activities on which f repetitively depends. In our example, this set is $Rd(f) = \{f, g, j\}$. The projection of the traces of L_2 onto $Rd(f)$, denoted by $Pr_{Rd(f)}(L_2)$, results in the set: $\{\langle f, g, j, f, g, j, f, g, j, f, g, j, f, g\rangle, \langle f, g\rangle, \langle f, g\rangle\}$. This set is a child node of the root.

Note that we did approximate the yellow loop (Figure 4) using the repeating dependency. We project the log onto $Rd(f) = \{f, g, j\}$ because, as we mentioned when illustrating our hypothesis, we want to leave out all behavior outside of cycles.

Now, we remove from L_2 all the activities that appear in $Rd(f)$; the new "event log" is $L_2^{'} = \{\langle q, s, w, e, r, i, a, y, u, d, o, e, r, t, i, a, u, d, h, k, z, x, c\rangle, \langle q, w, s, e, r, y, i, u, a, d, o, e, i, h, r, a, y, u, d, o, e, i, r, y, a, u, d, l, n\rangle, \langle q, s, w, e, i, r, a, y, u, d, o, e, i, h, a, r, d, y, u, o, e, i, a, r, y, u, d, o, e, i, a, r, t, u, d, l, n\rangle\}$.

We proceed in a similar way with the event log $L_2^{'}$. We find the first repeated activity in any of its traces. In this example, the activity is e, when we start from the first trace. Although any other trace with repeated activities can be used. We compute $Rd(e)$ for $L_2^{'}$, resulting: $Rd(e) <= \{r, e, i, a, y, u, d, o\}$. Now, we project the traces of $L_2^{'}$ onto $Rd(e)$, resulting in the set $Pr_{Rd(e)}(L_2^{'}) = \{\langle e, r, i, a, y, u, d, o, e, r, i, a, u, d\rangle, \langle e, r, y, i, u, a, d, o, e, i, r, a, y, u, d, o, e, i, r, y, a, u, d\rangle, \langle e, i, r, a, y, u, d, o, e, i, a, r, d, y, u, o, e, i, a, r, y, u, d, o, e, i, a, r, u, d\rangle\}$. The projection of the log onto $Rd(e)$ is another child of the root node. In this case, we approximate the pink cycle using the set $Rd(e) = \{r, e, i, a, y, u, d, o\}$.

Now, we remove all activities that appear in $Rd(e)$ from the log $L_2^{'}$. The resulting "event log" is $L_2^{''} = \{\langle q, s, w, t, h, k, z, x, c\rangle, \langle q, w, s, h, l, n\rangle, \langle q, s, w, h, t, l, n\rangle\}$. We call this set $L_2^{''}$. In $L_2^{''}$ there are no repeating activities, therefore it represents the activities from WF-net that are not included in a cycle. $L_2^{''}$ is the last descendant from the root node.

The following algorithm summarizes the procedure described.

Algorithm 1. Find Child Nodes

Input: L ◁ Set of traces contained in the node to expand
Output: Child Nodes of L

```
1.  $childNodes \leftarrow \emptyset$
2.  $auxLog \leftarrow L$
3.  while $auxLog$ $has$ $repeated$ $activities$ do:
4.      $repAct \leftarrow findFirstRep.(auxLog)$
5.      $dR \leftarrow computeRepetitive.(repAct)$
6.      $projectedLog \leftarrow Pr_{dR}(auxLog)$
7.      $childNodes \leftarrow childNodes \cup \{projectedLog\}$
8.      $auxLog \leftarrow Pr_{A \setminus dR}(auxLog)$    ◁ $A$ is the set of activities
9.  end while
10. $childNodes \leftarrow childNodes \cup \{auxLog\}$
11. return $childNodes$
```

The $findFirstRep$ function returns the first repeated activity in a trace of $auxLog$, and is stored in $repAct$. The $computeRepetitive$ function returns the set of activities that $repAct$ repetitively depends on and is stored in dR.

Once the root has been expanded, the rest of the tree is built by expanding the children by width traversal. Leaf nodes contain a set of traces, such that all activities that can be retrieved via the $findFirstRep$ function lead to a set of traces already contained in another node.

The black subgraph in Figure 6 shows the tree built from L_2. The gray nodes would be the children of the leaf nodes; however, as mentioned, these contain sets of traces that have already been found before. Therefore they are not considered.

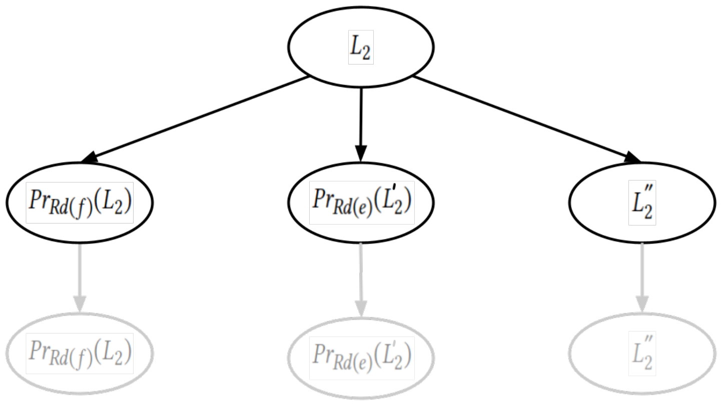

Figure 6: Tree generated using algorithm 1, from the event log L_2.

3.2.4. Step 2: Finding concurrency

Once the tree is built, we traverse its nodes in depth-first order. Upon reaching a node, the concurrency relations between activities in the traces of that node are extracted as follows:

- For *leaf nodes*, we transform the log contained in the node to a non-repetitive event log (Definition 3.2). The resulting log fulfills the assumptions of the Agrawal algorithm. So, we apply the Agrawal algorithm to this log, and the pairs of concurrent activities returned by the algorithm are added to the set of concurrent relations.
- For *internal nodes*, there is a slight change. Consider an internal node named *nd*. First, the non-repeating event log is obtained from the log contained in node *nd*. We applied the Agrawal algorithm to this non-repeating log, but since we are not making any assumptions about the rate of completeness of the log, it is impossible to guarantee that the found concurrency relations are all WF-net concurrences. That is because we are inferring beyond what is recorded. However, we apply the following *heuristic rule* to decide if a concurrency relationship returned by Agrawal's algorithm holds in the set of concurrency relationships delivered by our oracle.
  If Agrawal's algorithm returns that x and y are concurrents, we must consider the following cases:
  1. x and y belong to the set of activities of one of the child nodes of *nd*.
  2. x belongs to the set of activities of one child node of *nd*, and y belongs to the set of activities of another different child node of *nd*.
  3. x belongs to one of the child nodes of *nd* and y to none.
  4. neither x nor y belongs to any of the children nodes of *nd*.

  In the first case, the concurrency between x and y will be preserved only if it was found when the child node was parsed. If there are concurrency between x and y, it should have been observed when executing the innermost cycle (deepest node) that contains x and y. If it was not observed when analyzing the inner cycle, there is no reason why it could be observed in a larger cycle.
  In the second case, concurrency is preserved if and only if there is some explicitly observed concurrency (according to definition 2.13) between one of the activities of one node and an activity of the other node. This is because two inner cycles of a third cycle can be concurrent with each other, but they can also be sequential. To ensure that we are not adding false concurrency relationships (which actually are sequential relationships), we require that at least one concurrency relationship has been observed between the activities in the inner loops.
  In cases 3 and 4, concurrency is always preserved. The reason is that they are activities inside a cycle with nested cycles, but those activities are in the "non-repetitive" part of the cycle; that is, they are not inside any of the nested cycles. Therefore, the fact that they appeared in reverse order, as required by Agrawal's algorithm, is indicative that they are, in fact, concurrent.

Following our example, the first node visited contains the traces $\{\langle f, g, j, f, g, j, f, g, j, f, g, j, f, g\rangle, \langle f, g\rangle, \langle f, g\rangle\}$. Applying the described procedure, no concurrency relationship was found there, which is consistent with the model.

The second node visited contains the traces $\{\langle e, r, i, a, y, u, d, o, e, r, i, a, u, d\rangle, \langle e, r, y, i, u, a, d, o, e, i, r, a, y, u, d, o, e, i, r, y, a, u, d\rangle, \langle e, i, r, a, y, u, d, o, e, i, a, r, d, y, u, o, e, i, a, r, y, u, d, o, e, i, a, r, u, d\rangle\}$. Applying the procedure, it follows that $Conc1 = \{(r, i), (u, a), (i, y), (r, a), (u, d), (y, a), (y, d)\}$ are concurrency relations, which is consistent with the model.

The third node visited contains the traces $Conc2 = \{\langle q, s, w, t, h, k, z, x, c\rangle, \langle q, w, s, h, l, n\rangle, \langle q, s, w, h, t, l, n\rangle\}$. Applying the procedure, it follows that $\{(s, w), (t, h)\}$ are concurrency relations, which is consistent with the model.

The last node visited contains the original event log. Following the described procedure, it follows that $Conc3 = \{(e, g), (f, u), (f, a), (u, g), (f, o), (o, g), (j, u), (f, y), (d, g), (f, r), (i, g), (f, e), (f, d), (f, I), (r, j), (a, g), (r, g), (j, e)\}$ are concurrency relations, which is consistent with the model.

Therefore, the concurrences returned by our oracle are $Conc = Conc1 \cup Conc2 \cup Conc3$. Only the red ones are found by the α-oracle.

4. Implementation and Tests

The procedures derived from the proposed method have been implemented as a software tool in Phyton. It is available at https://www.websysmex.online/#/Cesar. The experiments have been performed on eight test cases where artificial logs are obtained from Petri nets with diverse structures; the logs are obtained by executing the modes in the PIPE tool [22].

The reason for using artificial event logs instead of real-life ones is that it is possible to compare the effectiveness of the method to known concurrency relationships in the test PN, which are unknown in real-life event logs. Petri nets and their corresponding event logs are available too at the same site. In the tests, we use event logs with rates of completeness of 30%, 40%, and 50%, approximately. The results of the tests are compared to those of the α-oracle.

Below the results of the method are shown for the eight event logs: O is the set of pairs of concurrent relations returned by our oracle, O_α is the set of pairs of concurrent relations computed by the α-oracle, and $N_{||}$ is the set of concurrent relations in the Petri net N.

Figure 7 shows $\frac{|O \cap N_{||}|}{|N_{||}|}$ for each of the eight Petri nets and the three rates of completeness; in other words, the percentage of the actual concurrency relationships of the Petri net that were found by our method. Analogously, Figure 8 shows $\frac{|O_\alpha \cap N_{||}|}{|N_{||}|}$. As expected, the higher the rate of completeness of the event log, the higher the percentage of revealed concurrency relationships.

The results achieved by our proposal show that it can extract significantly more concurrency relationships than those explicitly recorded.

However, in no case were they able to reveal the total attendance. Furthermore, the percentage of concurrency relationships found is very similar to the completeness rate for some nets.

Extracting unrecorded concurrency relations would be trivial if overgeneralization did not matter. However, discovering precise models requires that the oracle return as few false concurrency relationships as possible.

Figure 9 shows $\frac{|O \cap N_{||}|}{|O|}$, the percentage of concurrency relations of the PN extracted by our oracle concerning the total number of relations that the oracle determined as concurrences.

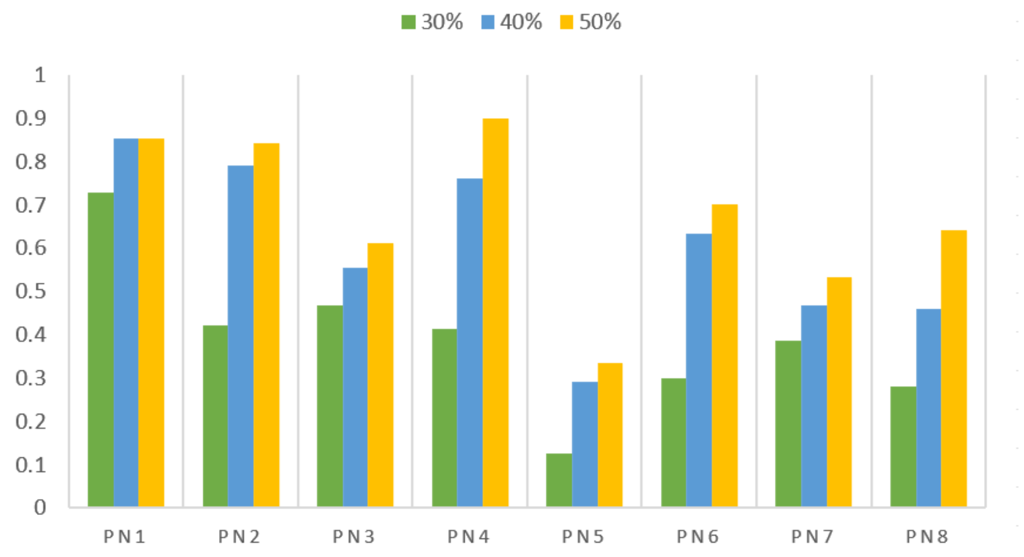

Figure 7: Percentage of concurrency relationships found for the proposed oracle.

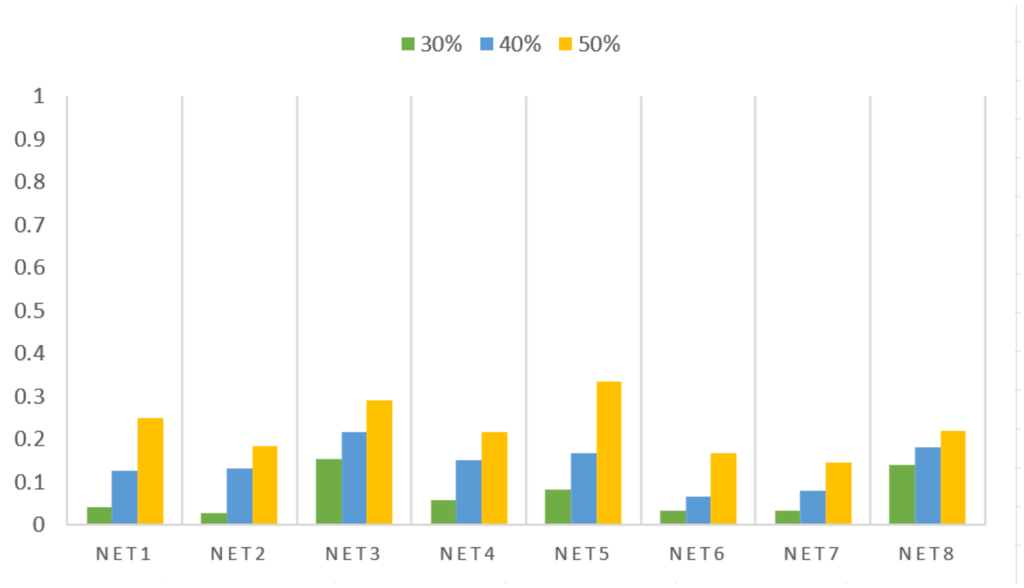

Figure 8: Percentage of concurrency relationships found for the α-oracle.

Figure 10 shows the same rate for the α-oracle ($\frac{|O_\alpha \cap N_{||}|}{|O_\alpha|}$).

Our proposal achieved results above 90% for most of the nets, which we consider favorable. However, for Petri net 6, it was found that almost 30% of the relationships that the oracle ”said” were concurrent were not. Therefore, in this case, using our oracle could lead to models with low precision. It is also striking that, in some cases, the higher the completeness rate, the more

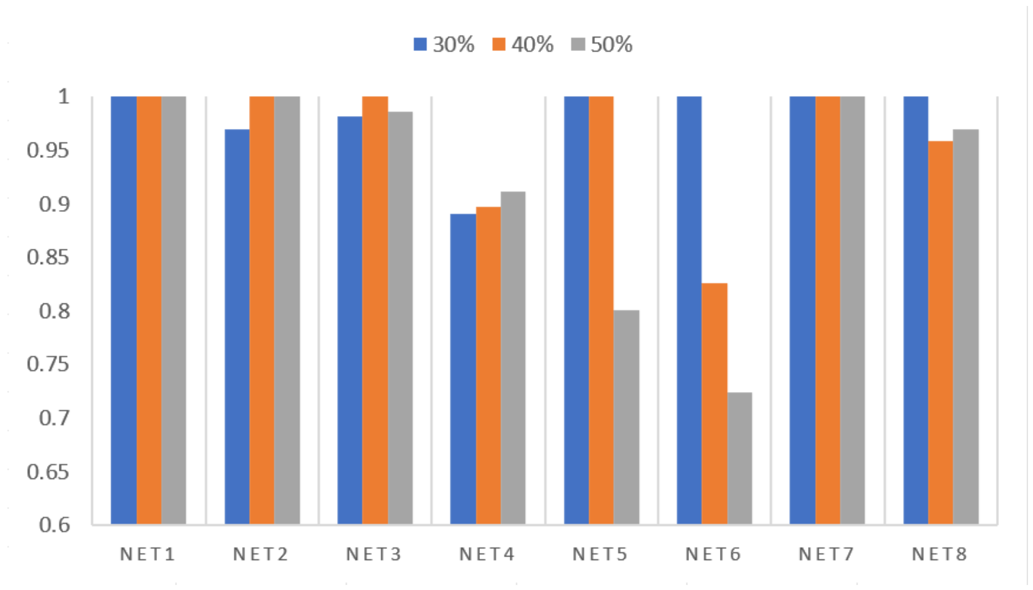

Figure 9: Percentage of concurrency relationships found by the proposed oracle that are concurrent relationships in the Petri net

false matches are obtained. As expected, the alpha oracle gets fewer errors. All α-oracle errors are associated with short loops.

5. Conclusions

This paper proposed a novel approach to building concurrency oracles based on discovering repetitive patterns from the event log. It has been shown in the tests that although the implemented oracle cannot determine all the concurrences, it can infer significantly more concurrency relationships than those straightforwardly determined from the event log. These results are encouraging and show that a concurrency oracle can be helpful when dealing with incomplete logs.

However, a few issues remain before it can be used in real logs. Mainly, the surplus of false concurrency relationships should be mitigated. Heuristic rules like that in step two of our proposal can reduce spurious concurrences. This approach has the advantage of not imposing constraints on the event log; however, for the same reason, it will not be possible to guarantee that all the concurrences found are indeed in the net.

Another approach to solve this issue is to constrain the class of WFNs to be dealt with to find a ”minimum behavior” that the log must hold to ensure that all the concurrency relationships determined by the oracle are true. Although we aim to make the oracle usable in actual environments, we prefer the first option. Our current research goes in that direction.

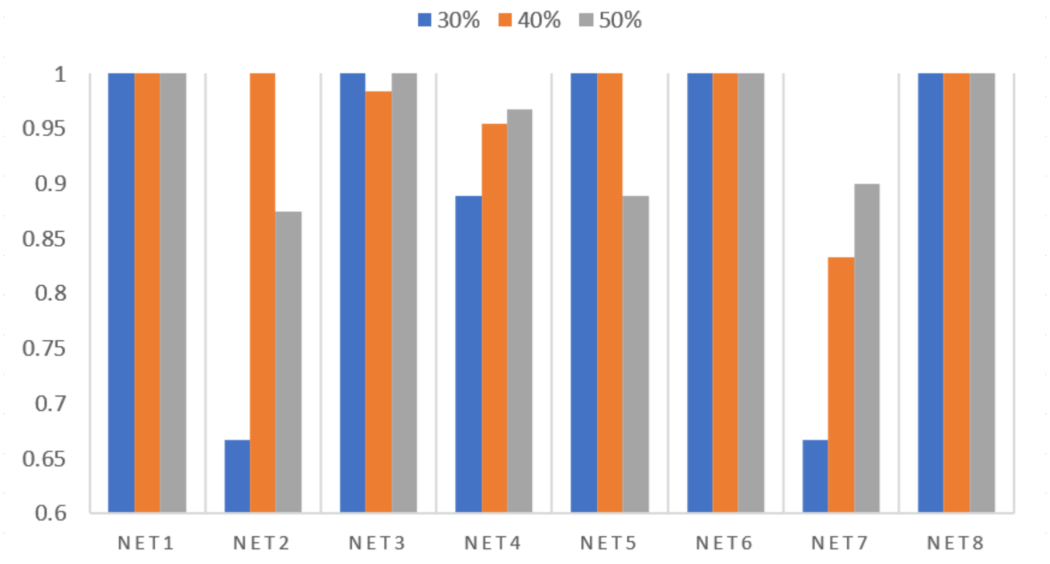

Figure 10: Percentage of concurrency relationships found by the α-oracle that are concurrent relationships in the Petri net.

References

[1] A. Armas-Cervantes, M. Dumas, M. L. Rosa, A. Maaradji, Local concurrency detection in business process event logs, ACM Transactions on Internet Technology (TOIT) 19 (2019) 1–23.

[2] S. J. Leemans, S. J. van Zelst, X. Lu, Partial-order-based process mining: a survey and outlook, Knowledge and Information Systems 65 (2023) 1–29.

[3] R. Bergenthum, J. Desel, R. Lorenz, S. Mauser, Synthesis of petri nets from finite partial languages, Fundamenta Informaticae 88 (2008) 437–468.

[4] R. Bergenthum, Synthesizing petri nets from hasse diagrams, in: Business Process Management: 15th International Conference, BPM 2017, Barcelona, Spain, September 10–15, 2017, Proceedings 15, Springer, 2017, pp. 22–39.

[5] R. Bergenthum, Prime miner-process discovery using prime event structures, in: 2019 International Conference on Process Mining (ICPM), IEEE, 2019, pp. 41–48.

[6] M. Dumas, L. García-Bañuelos, Process mining reloaded: Event structures as a unified representation of process models and event logs, in: Application and Theory of Petri Nets and Concurrency: 36th International Conference, PETRI NETS 2015, Brussels, Belgium, June 21-26, 2015, Proceedings 36, Springer, 2015, pp. 33–48.

[7] W. Van der Aalst, T. Weijters, L. Maruster, Workflow mining: Discovering process models from event logs, IEEE transactions on knowledge and data engineering 16 (2004) 1128–1142.

[8] L. Wen, W. M. Van Der Aalst, J. Wang, J. Sun, Mining process models with non-free-choice constructs, Data Mining and Knowledge Discovery 15 (2007) 145–180.

[9] A. K. A. de Medeiros, W. M. van der Aalst, A. Weijters, Workflow mining: Current status

and future directions, in: On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003. Proceedings, Springer, 2003, pp. 389–406.

[10] H. Ponce-de León, C. Rodríguez, J. Carmona, K. Heljanko, S. Haar, Unfolding-based process discovery, in: Automated Technology for Verification and Analysis: 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings 13, Springer, 2015, pp. 31–47.

[11] J. E. Cook, A. L. Wolf, Event-based detection of concurrency, ACM SIGSOFT Software Engineering Notes 23 (1998) 35–45.

[12] W. Van Der Aalst, A. Adriansyah, A. K. A. De Medeiros, F. Arcieri, T. Baier, T. Blickle, J. C. Bose, P. Van Den Brand, R. Brandtjen, J. Buijs, et al., Process mining manifesto, in: Business Process Management Workshops: BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I 9, Springer, 2012, pp. 169–194.

[13] J. Desel, J. Esparza, Free choice Petri nets, 40, Cambridge university press, 1995.

[14] W. Van Der Aalst, Process mining: data science in action, volume 2, Springer, 2016.

[15] N. Tax, N. Sidorova, W. M. van der Aalst, Discovering more precise process models from event logs by filtering out chaotic activities, Journal of Intelligent Information Systems 52 (2019) 107–139.

[16] R. Conforti, M. La Rosa, A. H. ter Hofstede, Filtering out infrequent behavior from business process event logs, IEEE Transactions on Knowledge and Data Engineering 29 (2016) 300–314.

[17] T. Tapia-Flores, E. Rodríguez-Pérez, E. López-Mellado, Discovering process models from incomplete event logs using conjoint occurrence classes., in: ATAED@ petri nets/ACSD, 2016, pp. 31–46.

[18] D. Reißner, A. Armas-Cervantes, M. La Rosa, Generalization in automated process discovery: A framework based on event log patterns, arXiv preprint arXiv:2203.14079 (2022).

[19] R. Agrawal, D. Gunopulos, F. Leymann, Mining process models from workflow logs, in: Advances in Database Technology—EDBT'98: 6th International Conference on Extending Database Technology Valencia, Spain, March 23–27, 1998 Proceedings 6, Springer, 1998, pp. 467–483.

[20] G. Juhás, R. Lorenz, J. Desel, Can i execute my scenario in your net?, in: Applications and Theory of Petri Nets 2005: 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005. Proceedings 26, Springer, 2005, pp. 289–308.

[21] T. Tapia-Flores, E. López-Mellado, A. P. Estrada-Vargas, J.-J. Lesage, Discovering petri net models of discrete-event processes by computing t-invariants, IEEE Transactions on Automation Science and Engineering 15 (2017) 992–1003.

[22] P. Bonet, C. M. Lladó, R. Puijaner, W. J. Knottenbelt, et al., Pipe v2. 5: A petri net tool for performance modelling, in: Proc. 23rd Latin American Conference on Informatics (CLEI 2007), 2007.