Mining local process models

Tax, N.; Sidorova, N.; Haakma, R.; van der Aalst, W.M.P.

Document Version
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Mining local process models

CrossMark

Niek Tax [a,b,*], Natalia Sidorova [a], Reinder Haakma [b],
Wil M.P. van der Aalst [a]

[a] *Eindhoven University of Technology, P.O. Box 513, Eindhoven, The Netherlands*
[b] *Philips Research, Prof. Holstlaan 4, 5665 AA Eindhoven, The Netherlands*

HIGHLIGHTS

- This paper presents a new data mining method called local process model (LPM) mining.
- LPM mining extends sequential pattern mining techniques to more complex patterns.
- LPM mining enables process mining of noisy data by focusing on local structures.

ARTICLE INFO

ABSTRACT

In this paper we describe a method to discover frequent behavioral patterns in event logs. We express these patterns as *local process models*. Local process model mining can be positioned in-between process discovery and episode/sequential pattern mining. The technique presented in this paper is able to learn behavioral patterns involving sequential composition, concurrency, choice and loop, like in process mining. However, we do not look at start-to-end models, which distinguishes our approach from process discovery and creates a link to episode/sequential pattern mining. We propose an incremental procedure for building local process models capturing frequent patterns based on so-called process trees. We propose five quality dimensions and corresponding metrics for local process models, given an event log. We show monotonicity properties for some quality dimensions, enabling a speedup of local process model discovery through pruning. We demonstrate through a real life case study that mining local patterns allows us to get insights in processes where regular start-to-end process discovery techniques are only able to learn unstructured, flower-like, models.

1. Introduction

Process mining aims to extract novel insight from event data [1]. Process discovery, the task of discovering a process model that is representative for the set of event sequences in terms of start-to-end behavior, i.e. from the start of a case till its termination, plays a prominent role in process mining. Many process discovery algorithms have been proposed and applied to a variety of real life cases (see [1] for an overview). A different perspective on mining patterns in event

---

sequences can be found in the data mining field, where the episode mining [2] and sequential pattern mining [3] subfields focus on finding frequent patterns that are local, not necessarily describing the whole event sequences from start to end. Episode mining and sequential pattern mining have e.g. been used to analyze telecommunication networks [2], web navigational logs [2,4], and retail sales transactions [5].

Sequential pattern mining and episode mining are limited to the discovery of *sequential orderings* or *partially ordered sets* of events, while process discovery methods aim to discover a larger set of event relations, including sequential orderings, (exclusive) choice relations, concurrency, and loops, represented in process models such as Petri nets [6], BPMN [7], and process trees [8]. Process models that can be discovered with process discovery methods distinguish themselves from more traditional sequence mining methods like Hidden Markov Models [9] and Recurrent Neural Networks [10,11] in that process models can be visually represented and their visual representation can be used for communication between process stakeholders. However, process discovery is normally limited to the discovery of a model capturing the behavior of process instances as a whole, and not local patterns within instances. Our goal is to develop methods allowing to mine *local* process models positioned in-between simple patterns (e.g. subsequences) and start-to-end models. Local process models focus on a subset of the process activities and describe some behavioral pattern that occurs frequently within event sequences. Such local process models cannot be discovered by using standard techniques.

Imagine a sales department where multiple sales officers perform four types of activities: (A) register a call for bids, (B) investigate a call for bids from the business perspective, (C) investigate a call for bids from the legal perspective, and (D) decide on participation in the call for bid. The event sequences (Fig. 1(a)) contain the activities performed by one sales officer throughout the day. The sales officer works on different calls for bids and not necessarily performs all activities for a particular call himself. Applying discovery algorithms, like the Inductive Miner [12], yields models allowing for any sequence of events (Fig. 1(c)). Such "flower-like" models do not give any insight in typical behavioral patterns. When we apply any sequential pattern mining algorithm using a threshold of six occurrences, we obtain the seven length-three sequential patterns depicted in Fig. 1(d) (results obtained using the SPMF [14] implementation of the PrefixSpan algorithm [13]). However, the data contains a frequent non-sequential pattern where a sales officer first performs A, followed by B and a C in arbitrary order (Fig. 1(b)). This pattern cannot be found with existing process discovery and sequential pattern mining techniques. The two numbers shown in the transitions (i.e., rectangles) represent (1) the number of events of this type in the event log that fit this local process model and (2) the total number of events of this type in the event log. For example, 13 out of 19 events of type C in the event log fit transition C, which are indicated in bold in the log in Fig. 1(a). Underlined sequences indicate non-continuous instances, i.e. instances with non-fitting events in-between the events forming the instance of the local process model.

In this paper we describe a method to extract frequently occurring *local process models*, allowing for choice, concurrency, loops, and sequence relations. We leverage process trees [15] to search for local process models, and describe a way to recursively explore candidate process trees up to a certain model size. For convenience, we often use the Petri net representations for process trees. In fact, results can also be visualized as BPMN [7], EPC [16], UML activity diagram [17], or UML statechart diagram [17]. We define five quality dimensions that express the degree of representativeness of a local process model with regard to an event log: *support*, *confidence*, *language fit*, *coverage*, and *determinism*. Based on quality metrics, we describe monotonicity properties over some quality dimensions and show how they can be used to make the recursive search over process trees more efficient.

The paper is organized as follows. Section 2 introduces the basic concepts used in this paper. Section 3 describes related work in the fields of process discovery and sequential pattern mining. Section 4 describes our local process model mining approach. Section 5 introduces quality dimensions and metrics for local process models and discusses monotonicity properties. Section 6 describes a local process model evaluation approach based on alignments. Section 7 shows the relevance of the proposed technique using two real life data sets and compares the results with the results obtained with several related techniques. Section 8 concludes the paper.

2. Preliminaries

In this section we introduce process modeling notations, including Petri nets, process trees, which are used in later sections of this paper.

X^* denotes the set of all sequences over a set X and $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ a sequence of length n; $\langle \rangle$ is the empty sequence and $\sigma_1 \cdot \sigma_2$ is the concatenation of sequences σ_1, σ_2. $\sigma \restriction Q$ is the projection of σ on Q, e.g. $\langle a, b, c, a, b, c \rangle \restriction_{\{a,c\}} = \langle a, c, a, c \rangle$. $\#_a(\sigma)$ denotes the number of occurrences of element a in sequence σ, e.g. $\#_a(\langle a, b, c, a \rangle) = 2$.

Definition 1 (*Applying Functions to Sequences*). A partial function $f \in X \nrightarrow Y$ can be lifted to sequences over X using the following recursive definition: (1) $f(\langle \rangle) = \langle \rangle$; (2) for any $\sigma \in X^*$ and $x \in X$:

$$f(\sigma \cdot \langle x \rangle) = \begin{cases} f(\sigma) & \text{if } x \notin dom(f), \\ f(\sigma) \cdot \langle f(x) \rangle & \text{if } x \in dom(f). \end{cases}$$

We assume the set of all *process activities* Σ_L to be given. An *event* e is the occurrence of an activity $e \in \Sigma_L$. We call a sequence of events $t \in \Sigma_L{}^*$ a *trace*. An *event log* $L \in \mathbb{N}^{\Sigma_L{}^*}$ is a finite multiset of traces. For example, the event log $L = [\langle a, b, c \rangle^2, \langle b, a, c \rangle^3]$ consists of 2 occurrences of trace $\langle a, b, c \rangle$ and three occurrences of trace $\langle b, a, c \rangle$. We lift the sequence projection to the multisets of sequences in the standard way. For example, for the log L given above $L \restriction_{\{a,c\}} = [\langle a, c \rangle^5]$. We lift the number of occurrences in a sequence to multisets of sequences in the standard way, for example, $\#_a(L) = 5$.

Petri nets are directed bipartite graphs consisting of transitions and places, connected by arcs. Transitions represent activities, while places represent the enabling conditions of
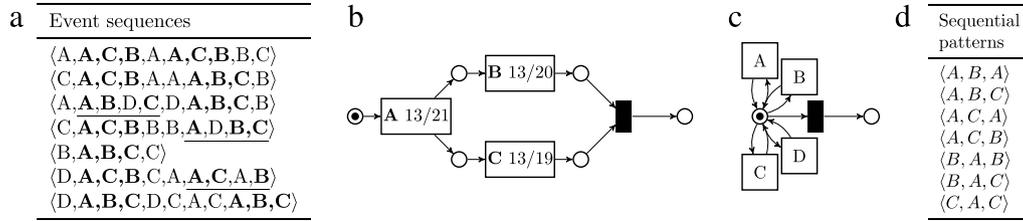
Fig. 1 – (a) A log L of sales officer event sequences with highlighted instances of the frequent pattern. (b) The local process model showing frequent behavior in L. (c) The Petri net discovered on L with the Inductive Miner algorithm [12]. (d) The sequential patterns discovered on L with the PrefixSpan algorithm [13].

transitions. Labels are assigned to transitions to indicate the type of activity that they model. A special label τ is used to represent invisible transitions, which are only used for routing purposes and not recorded in the execution log.

Definition 2 (*Labeled Petri Net*). A *labeled Petri net* $N = \langle P, T, F, \Sigma_M, \ell \rangle$ is a tuple where P is a finite set of places, T is a finite set of transitions such that $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs, called the flow relation, Σ_M is a finite set of labels representing activities, with $\tau \notin \Sigma_M$ being a label representing invisible events, and $\ell : T \to \Sigma_M \cup \{\tau\}$ is a labeling function that assigns a label to each transition.

For a node $n \in (P \cup T)$ we use $\bullet n$ and $n \bullet$ to denote the set of input and output nodes of n, defined as $\bullet n = \{n' | (n', n) \in F\}$ and $n \bullet = \{n | (n, n') \in F\}$.

A state of a Petri net is defined by its *marking* $M \in \mathbb{N}^P$ being a multiset of places. A marking is graphically denoted by putting $M(p)$ tokens on each place $p \in P$. A pair (N, M) is called a marked Petri net. State changes occur through transition firings. A transition t is enabled (can fire) in a given marking M if each input place $p \in \bullet t$ contains at least one token. Once a transition fires, one token is removed from each input place of t and one token is added to each output place of t, leading to a new marking M' defined as $M' = M - \bullet t + t \bullet$. A firing of a transition t leading from marking M to marking M' is denoted as $M \xrightarrow{l(t)} M'$. $M_1 \xrightarrow{l(\sigma)} M_2$ indicates that M_2 can be reached from M_1 through a firing sequence $\sigma' \in \Sigma_M^*$.

Often it is useful to consider a Petri net in combination with an initial marking and a set of possible final markings. This allows us to define the language accepted by the Petri net and to check whether some behavior is part of the behavior of the Petri net (can be replayed on it).

Definition 3 (*Accepting Petri Net*). An *accepting Petri net* is a triple $APN = (N, M_0, MF)$, where N is a labeled Petri net, $M_0 \in \mathbb{N}^p$ is its initial marking, and $MF \subseteq \mathbb{N}^p$ is its set of possible final markings, such that $\forall_{M_1, M_2 \in MF} M_1 \not\subseteq M_2$. A sequence $\sigma \in T^*$ is called a *trace* of an accepting Petri net APN if $M_0 \xrightarrow{\sigma} M_f$ for some final marking $M_f \in MF$. The *language* $\mathcal{L}(APN)$ of APN is the set of all its traces.

Fig. 2 shows an example of an accepting Petri net. Circles represent places and rectangles represent transitions. Invisible transitions (labeled τ) are depicted as black rectangles. Places that belong to the initial marking contain a token and places belonging to a final marking contain a bottom right label f_i with i a final marking identifier, or are simply marked
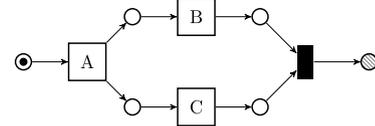

Fig. 2 – Example of an accepting Petri net.

as ◎ in case of a single final marking. The language of this accepting Petri net is $\{\langle A, B, C \rangle, \langle A, C, B \rangle\}$.

A different process representation is a process tree [15]. Process trees can only model sound (deadlock-free and livelock-free) processes. The recursive definition of process trees make them a convenient representation to iteratively expand process models into larger process models.

Definition 4 (*Process Tree*). Let $A \in \Sigma_M$ be a finite set of activities with $\tau \notin \Sigma_M$. $\bigoplus = \{\to, \times, \wedge, \circlearrowleft\}$ is the set of *process tree* operators.

- if $a \in \Sigma_M \cup \{\tau\}$ then $Q = a$ is a process tree.
- if Q_1, Q_2 are process trees, and $\oplus \in \bigoplus$, then $Q = \oplus(Q_1, Q_2)$ is a process tree.

A *process tree* is a tree structure consisting of operator and activity nodes, such that each leaf node is an activity node and each non-leaf node is an operator node.

The *loop* operator ($\circlearrowleft$) has two child nodes, with the first child the "do" part and the second child the "redo" child. Process tree $p_1 = \circlearrowleft (a, b)$ accepts language $\mathcal{L}(p_1) = \{\langle a \rangle, \langle a, b, a, \rangle, \langle a, b, a, b, a \rangle, \ldots\}$.

The *sequence* operator ($\to$) has two children, such that the first child is executed prior to execution of the second child. The language of process tree $p_2 = \to (a, b)$ is $\mathcal{L}(p_2) = \{\langle a, b \rangle\}$.

The *exclusive choice* operator ($\times$) has two children, indicating that either the first or the second child will be executed, but not both. The language of process tree $p_3 = \times(a, b)$ is $\mathcal{L}(p_3) = \{\langle a \rangle, \langle b \rangle\}$.

The *concurrency* operator ($\wedge$) has two children, indicating that the both children will be executed in parallel. Process tree $p_4 = \wedge(\to (a, b), \to (c, d))$ accepts language $\mathcal{L}(p_4) = \{\langle a, b, c, d \rangle, \langle a, c, b, d \rangle, \langle a, c, d, b \rangle, \langle c, a, b, d \rangle, \langle c, a, d, b \rangle, \langle c, d, a, b \rangle\}$.

Let $p_5 = \times(\circlearrowleft (a, b), \wedge(c, d))$. Fig. 3 shows the graphical representation of p_5. Because the do and redo children of a loop operator can be executed an arbitrary number of times, process trees containing a loop operator have a language of infinite size.

We define *n-restricted language*, consisting of all language traces of at most length $n \in \mathbb{N}$, as $\mathcal{L}_n = \{t \in \mathcal{L} \mid |t| \leq$
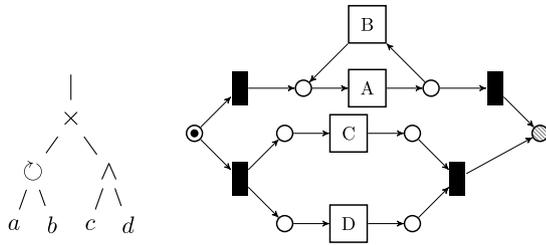
Fig. 3 – Graphical representation of process tree $\times(\circlearrowleft(a, b), \wedge(c, d))$**.**

$n\}$. The n-restricted language is guaranteed to be of finite size, independent of the operators in the process tree. The 5-restricted language of the process tree p_5 is $\mathfrak{L}_5(p_5) = \{\langle a\rangle, \langle a, b, a\rangle, \langle a, b, a, b, a\rangle, \langle c, d\rangle, \langle d, c\rangle\}$.

3.   Related work

ProM's Episode Miner [18] is a method that can be considered to be in-between episode mining and process mining, as it discovers a collection of patterns from an event log where each pattern consists of partial order constructs. However, contrary to the technique that we describe in this paper, ProM's Episode Miner does not support loop and exclusive choice constructs and is not easily extensible to include new types of constructs.

Lu et al. propose a method called Post Sequential Patterns Mining (PSPM) [19] that takes a input a set of sequential patterns and post-processes them into a single graph consisting of sequential and exclusive choice constructs, which they call a Sequential Pattern Graph (SGP) [20]. A later extension by Lu et al. adds the capability to mine concurrency relations [21]. An SGP can be discovered from an event log by first applying any existing sequential pattern mining algorithm followed by PSPM on the discovered set of sequential patterns. The reliance of PSPM methods on the output of sequential pattern mining techniques can also be considered to be a drawback of the approach. When two activities A and B are in parallel then both the orderings $\langle A, B\rangle$ and $\langle B, A\rangle$ will be present in the complete log. However, if one of the two orderings is more frequent than the other due to chance, one of the two orderings might not reach the support set in the sequential pattern mining, making it impossible for PSPM methods to discover the concurrency of A and B. A more fundamental distinction between PSPM and Local Process Model (LPM) discovery is that PSPM merges all relations into one single pattern while LPM discovery aims at discovering a collection of patterns. Merging separate patterns into a single graph could result in one single overgeneralizing graph. For example in the case of log $L = [\langle b, a, c\rangle^{100}, \langle d, a, e\rangle^{100}]$, sequential pattern mining techniques will find two sequential patterns: b, a, c and d, a, e. Merging them into a single graph where a is followed by either c or e and is preceded by either b or d results in the loss the long term dependency where b already determines the occurrence of a c after the a.

Jung et al. [22] describe a method to mine frequent patterns from a collection of process models by transforming each business process to a vector format and then applying agglomerative clustering. Diamantini et al. [23,24] take a similar approach, but apply graph clustering techniques instead of a traditional clustering approach. These techniques differ from LPM discovery as they take as input a set of process models instead of an event log. However, in many situations there are no existing process models available and, as shown in the introduction, it is not always possible to mine structured process models from an event log using process discovery techniques.

In later work, Diamantini et al. [25] describe a method to mine frequent patterns in process model notation through a two step approach. First each trace from the event log is transformed into a so called instance graph [26], which is graph representation of a trace that shows which steps in the trace are performed sequentially and which steps are performed in parallel (i.e. overlapping in time). In the second step they apply a graph clustering technique to obtain frequent subgraphs from this set of instance graphs. However, since instance graphs are limited to sequential and parallel constructs, other process constructs, such as choices and loops, cannot be discovered with the approach described in Diamantini et al. [25], while they can be discovered with LPM discovery.

The techniques developed in the area of trace clustering [27–30] are related to LPM discovery in the sense that both aim to enable extraction of process insight from event logs where the process is too unstructured for existing process discovery techniques to find a structured process model. Trace clustering techniques aim to solve this by clustering similar traces together to prevent mixing different usage scenarios into one single unstructured process model. Trace clustering techniques work well when the original event log does not originate from one single process, but in fact originates from multiple processes. However, not in all types of complex and flexible event data there is a cluster tendency in the data. An example for such non-clusterable event data can be found in the log shown in Fig. 1(a), where no clustering over the traces would enable the discovery of the frequent pattern shown in Fig. 1(b). The traces in the log have large parts of randomness *within* the traces, while trace clustering helps for cases where there is a large degree of variety *between* traces.

Declarative process models, such as Declare [31], define the allowed behavior through constraints that must be respected while carrying out the process. This contrasts procedural process models, which are dominant in the process discovery field and specify all possible orderings of events explicitly. Two examples of process discovery approaches that generate declarative process models are the DPIL Miner [32] and the Declare Miner [33]. Both approaches specify a set of *rule templates* that consists of two activity variables and their relation. An example of such a template is *sequence*(a, b), indicating that some activity a is followed by b. Concrete rules are extracted from the event log based on this template-based search. However, since the rule templates are limited to relations between two activities, more complex relations between three or more activities cannot be discovered. Imagine that for some event log a declarative process discovery method finds two relations: *sequence*(a, b) and *sequence*(b, c), indicating that both the occurrences of activity b after a and the occurrences of activity c after b meet some support threshold. The binary relations *sequence*(a, b) and *sequence*(b, c) combined

do not imply a tertiary relation equivalent to process tree $\rightarrow (a, \rightarrow (b, c))$, since it could be the case that specifically those occurrences of b that are preceded by a are rarely followed by c. The LPM discovery approach discussed in this paper enables discovery of relations between three or more activities.

Hybrid process discovery [34] aims at discovering a process model that consists partially of procedural process model constructs and partially of declarative process model constructs. Existing hybrid process discovery approaches consist of a first step where activities are separated into a group of structured activities and a group of unstructured activities, based on the number of unique predecessors and successors of an activity. However, some activities, such as activity a in the event log of Fig. 1(a), are part of a frequent pattern, but also to occur as noise at random point in the traces. Such activities would be classified as noisy activities by existing hybrid process discovery approaches, resulting in this activity being modeled with binary declarative constructs.

The Fuzzy Miner [35] is a process discovery technique developed to deal with complex and flexible process models. It connects nodes that represent activities with edges indicating follows relations, taking into account the relative significance of follows/precedes relations and allowing the user to filter out edges using a slider. However, the process models obtained using the Fuzzy Miner lack formal semantics, e.g. when a node has two or more outgoing edges, it is undefined whether this represents a choice, an exclusive choice, or parallel execution of the connected nodes.

We have described several techniques that are related in that sense that (i) they aim to enable mining of process insight from less structured processes on which traditional process discovery methods fail, or (ii) they aim to extract a collection of process models that each represent some subprocess. However, none of the existing techniques in category (i) is able to deal with event logs where some frequent patterns are surrounded by random events, as is the case in the event log in Fig. 1(a), and all of the existing methods in category (ii) either require a completely different type of input (a collection of graphs), or they support only a part of the constructs supported by the LPM discovery approach.

4.   Local process model discovery approach

A local process model (LPM) aims to describe frequent behavior in an event log in local, i.e. smaller, patterns, typically between three and five activity nodes. A LPM does not aim to describe all behavior in a trace completely, but instead aims to describe traces partially, focusing on frequently occurring patterns. A LPM LN represents behavior over Σ_M and accepts language $\mathfrak{L}(LN)$. The closure of the accepting language with respect to alphabet Σ_L is defined as $\overline{\mathfrak{L}}(LN, \Sigma_L) = \{\sigma \in \Sigma_L^* | \sigma \restriction_{\Sigma_M} \in \mathfrak{L}(LN)\}$.
Here, we introduce a local process model discovery approach that consists of four main steps:

(1) *Generation* Generate the initial set CM_1 (so $i = 1$) of candidate LPM in the form of process trees consisting of one leaf for each activity $a \in \Sigma_L$. Fig. 4 shows this set of elementary process trees for an event log over alphabet $\Sigma_L = \{a, b, \ldots, z\}$. Create selected set of selected LPMs $SM = \emptyset$.

Fig. 4 – Set of elementary process trees over $\Sigma_L = \{a, b, \ldots, z\}$.

(2) *Evaluation* Evaluate LPMs in current candidate set CM_i based on a set of quality criteria.
(3) *Selection* Based on evaluation results, a set $SCM_i \subseteq CM_i$ of candidate LPMs are selected. $SM = SM \cup SCM_i$. If $SCM_i = \emptyset$ or $i \geq max\_iterations$: stop.
(4) *Expansion* Expand SCM_i into a set of larger, expanded, candidate process models, CM_{i+1}. Goto step 2 using the newly created candidate set CM_{i+1}.

Expansion consists of the replacement of one of the leaf activity nodes a of the process tree by an operator node of one of the operator types, where one of the child nodes is the replaced activity node a and the other is a new activity node representing one of the activities $b \in \Sigma_L$. Fig. 5 shows the set of expansion operations to leaf node a, consisting of six possible expansions. Two types of expansion operations are defined for the sequence ($\rightarrow$) and the loop ($\circlearrowright$) operator types, as $\rightarrow (a, b) \not\equiv \rightarrow (b, a)$ and $\circlearrowright (a, b) \not\equiv \circlearrowright (b, a)$. However, only one expansion operation is needed for the $\wedge$ and $\times$ operators because of their symmetry ($\wedge(a, b) \equiv \wedge(b, a)$ and $\times(a, b) \equiv \times(b, a)$). We call $\wedge$ and $\times$ symmetrical operators.
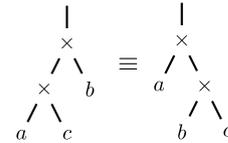
Expanding the first leaf node (a) of process tree $\times(a, b)$ with the $\times$ operator and some activity $c \in \Sigma_L$ results in the leftmost process tree depicted on the right, while applying the same expansion the second leaf node (b) of the same process tree results in a behaviorally equivalent process tree (as shown on the right). The same holds for expansion of $\wedge(a, b)$ with the $\wedge$ operator. Therefore, the number of expansions can be reduced further by restricting expansion of a leaf node that has a symmetrical operator as parent with the same symmetrical operator only to the rightmost child. This prevents unnecessary computation by generating both of the behaviorally equivalent trees shown on the right.

The number of possible expansion operations for a process tree P grows with the size of the alphabet of the event log $|\Sigma_L|$ and the number of activity nodes in P. This is easy to see, as each type of expansion operation can be applied to each activity node in P, leading to $6 \times |\Sigma_L|$ expansion operations per activity node. At every point in the expansion process, the number of activity nodes in the tree is equal to the number of expansion operations performed plus one, as each expansion operation adds one activity node to the process tree.

The local process model discovery procedure stops when no process tree in the current set of candidate process models meets the quality criteria, or, to guarantee termination, when a maximum number of expansion steps, *max_iterations*, is reached.

The approach of iteratively expanding, selecting, and expanding process trees described above is not theoretically
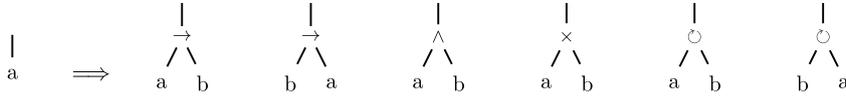
Fig. 5 – The set of process tree expansion operations to leaf node a, where $b \in \Sigma_L$.

limited to the set of operator nodes described above, and can easily be extended to other operators, such as an inclusive choice operator or a long-term dependency operator. Adding extra operators, however, comes with the price of increased computational complexity as it increases the number of ways to expand a process tree.

5. Quality criteria & metrics

Assume for now that we have an oracle function λ_{LN} that generates a segmentation of a given trace σ: $\lambda_{LN}(\sigma) = \gamma_0 \xi_1 \gamma_1 \xi_2 \dots \xi_k \gamma_k$, with $\xi_i \in \mathcal{L}(LN)$ and $\gamma_i \notin \mathcal{L}(LN)$, such that the number of events in $\{\xi_1, \dots, \xi_k\}$ is maximized: the higher the number of events in $\xi_i \in \mathcal{L}(LN)$ segments, the larger the share of trace σ explained by the LPM. A higher value k indicates more frequent presence of LPM execution traces in σ. $\lambda_{LN}^{k}(\sigma)$ denotes the number of $\xi \in \mathcal{L}(LN)$ segments in $\lambda_{LN}(\sigma)$. $\lambda_{LN}^{\xi}(\sigma)$ denotes the multiset of segments $\xi_i \in \mathcal{L}(LN)$. Here we discuss a number of quality criteria for LPM with regard to an event log.

Support Relates to the number of fragments in the event log that can be considered to be an instance of the LPM under evaluation. The rationale behind this quality dimension: an LPM whose execution traces are observed more frequently in the event log represents it better. We transform the count of pattern instances of LN in L into a $[0, 1)$-interval number through the following transformation:

$$support(LN, L) = \frac{\sum\limits_{\sigma \in L} \lambda_{LN}^{k}(\sigma)}{\left(\sum\limits_{\sigma \in L} \lambda_{LN}^{k}(\sigma)\right) + 1}.$$

Confidence An event fits an LPM when it is part of a segment $\xi \in \mathcal{L}(LN)$. The confidence of event type $e \in \Sigma_M$ in LPM LN given event log L, is the ratio of events of type a in L that fit LN:

$$confidence(a, L) = \frac{\sum\limits_{\sigma \in L} \#_a(\lambda_{LN}^{\xi}(\sigma))}{\#_a(L)}.$$

We use the harmonic mean to aggregate confidence values for individual activities to a single metric, as it is more sensitive to a single lower-than-average value than the geometric mean. We define the confidence of an LPM LN given an event log L to be the harmonic mean of the individual confidence scores of the event types of LN:

$$confidence(LN, L) = \frac{|\Sigma_M|}{\sum\limits_{a \in \Sigma_M} \frac{1}{confidence(a, L)}}.$$

Language fit Language fit expresses the ratio of the behavior allowed by the LPM that is observed in the event log. LPMs that allow for much more behavior than what is observed are likely to overgeneralize and therefore do not describe the

behavior in the event log well. The language fit of an LPM LN given log L is:

$$language\_fit(LN, L) = \frac{|\{\phi \in \mathcal{L}(LN) | \exists \sigma \in L : \phi \in \lambda_{LN}^{\xi}(\sigma)\}|}{|\mathcal{L}(LN)|}.$$

Since $|\mathcal{L}(LN)| = \infty$ in case LN contains a loop, $language\_fit(LN, L) = 0$ for any LN containing a loop. Restricting the language and the LPM instances to sequences of a bounded length allows us to approximate language fit for models with infinite size language. Language fit restricted to bound $n \in \mathbb{N}$ is defined as:

$$language\_fit_n(LN, L) = \frac{|\{\phi \in \mathcal{L}_n(LN) | \exists \sigma \in L : \phi \in \lambda_{LN}^{\xi}(\sigma)\}|}{|\mathcal{L}_n(LN)|}.$$
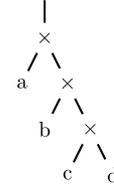
Determinism Flower-like process trees, like the one shown on the right, are not desirable as they provide little insight in what behavior we are likely to observe. Deterministic LPMs have more predictive value in with regard to future behavior. When the language of LPM LN contains traces if type $\sigma a \gamma_1$ and $\sigma b \gamma_2$, the continuation of the trace after observing prefix σ can be either a or b, leaving some uncertainty. LPMs with a high degree of certainty are preferable over LPMs with a low degree of certainty. A metric for the determinism quality dimension is dependent on the process model and not only on its language. Let $\mathcal{R}(LN)$ be a set of reachable states of an LPM LN. $\mathcal{W}_L : \mathcal{R}(LN) \to \mathbb{N}$ represents a function assigning the number of times a state is reached while replaying the fitting segments of log L on LN. $\mathcal{D} : \mathcal{R}(LN) \to \mathbb{N}$ represents a function assigning the number of transitions enabled in a certain state in LN. Determinism is defined as:

$$determinism(LN, L) = \frac{\sum\limits_{m \in \mathcal{R}(LN)} \mathcal{W}_L(m)}{\sum\limits_{m \in \mathcal{R}(LN)} . \mathcal{W}_L(m) \cdot \mathcal{D}(m)}.$$

Coverage Let LN be an LPM and L be an event log. Let $\#_*(L)$ denote the total number of events of event log L. Coverage is defined as the ratio of the number of events in the log after projecting the event log on the labels of LN to the number of all events in the log:

$$coverage(LN, L) = \frac{\#_*(L \restriction_{\Sigma_M})}{\#_*(L)}.$$

5.1. Local process model selection & ranking

The quality dimensions and metrics defined are used to select and rank local process models generated through the

recursive process tree exploration approach. Often, one is interested in multiple quality criteria at the same time. A high-support local process model that has a low determinism score (e.g., a small flower pattern) does not generate much process insight, while a deterministic pattern that has low support does not describe the behavior in the log very well. So it is possible to set thresholds per dimension. It is also useful to rank patterns according to a weighted average over the quality criteria. The appropriate weighting of the quality dimensions depends on the business questions and the situation at hand.

5.2. *Monotonicity properties & pruning*

Often one is not interested in local process models with a low support, confidence, or determinism. Setting a minimum threshold for these quality criteria allows us to prune away those parts of the search space where we know that expansions of a candidate local process model can never meet the criteria because of monotonicity, resulting in a speedup of the proposed recursive process tree exploration procedure. Pruning based on monotonicity is similar to the pruning performed in the well-known Apriori algorithm [36], and other algorithms inspired by the Apriori algorithm, such as [3].

Any expansion of process tree P where a leaf node $a \in P$ is replaced by subtree $\rightarrow (a, b)$, $\rightarrow (b, a)$, $\wedge(a, b)$, or $\circlearrowleft (a, b)$ for any $b \in \Sigma_L$ is guaranteed to be less frequent, i.e. has lower support, than P. The intuition behind this is that expansion put additional requirements of the alignments, possibly causing some fitting segments for a trace σ obtained by $\lambda_P^\xi(\sigma)$ to not fit the expansion of P. Therefore, when P does not meet support threshold $\min_{sup}$, its expansions of any activity node a into $\rightarrow (a, b)$, $\rightarrow (b, a)$, $\wedge(a, b)$, and $\circlearrowleft (a, b)$ can be pruned from the search space.

Process tree P is guaranteed to be at least as deterministic as its expansion where activity node $a \in P$ is replaced by subtree $\times(a, b)$ or $\wedge(a, b)$ for any $b \in \Sigma_L$. Therefore, when P does not meet determinism threshold $\min_{det}$, its expansions of any activity node a into $\times(a, b)$, and $\wedge(a, b)$ can be pruned from the search space.

6. Alignment-based evaluation of local process models

We now describe a way to define function λ_{LN}. We evaluate LPMs using Petri nets because of the rich set of analysis techniques available for Petri nets. Important for the definition of λ_{LN} is the notion of alignments [37], which aims to find a sequence of model firings starting at the initial marking and ending in a final marking that is an optimal approximation of the behavior in the event log. Alignments relate model traces and event log traces through a series of three types of moves: *synchronous moves*, *moves on model*, and *moves on log*. When an event in the event log trace can be performed in the process model, log and model can move *synchronously*. However, when a trace of the log does not fit the model, log and model cannot move synchronously from the start to the end of the trace. A *move on model* corresponds to a firing of a transition in the model that cannot be mapped to an event in the log. A *move on*

log corresponds to an event in the log that cannot be mapped to a transition firing in the model. Since both *moves on model* and *moves on log* are suboptimal behavior, they are often assigned certain costs such that the alignment will only chose to do *moves on model* or *moves on log* when these moves are unavoidable. *Moves on model* enable the unwanted behavior that a partial execution of the LPM can be identified as an LPM execution trace. To avoid this behavior, we use a version of alignments where moves on model on non-silent transitions are prohibited (infinite costs).

Alignments aim to match an event log trace with a single execution of a process model. However, an event log trace can contain more than one execution trace of an LPM. We modify the Petri net representation of the LPM such that we connect each final marking to the initial marking through a silent transition, allowing the alignment to relate a single trace to multiple executions of the model. Fig. 6(a) shows an example LPM and Fig. 6(b) shows the corresponding Petri net after transformation. We transform LPM $LN(N, M_0, MF)$ with $N = (P, T, F, \Sigma_M, \ell)$ into $LN_{BL}(N_{BL}, M_0, \{M_0\})$ with $N_{BL} = (P, T_{BL}, F_{BL}, \Sigma_M, \ell_{BL})$, such that:

- $T_{BL} = T \cup \{t_{bl_M} | M \in MF\}$,
- $F_{BL} = F \cup \{(p, t_{bl_M}) | M \in MF \wedge p \in M\} \cup \{t_{bl_M} | M \in MF \wedge p \in M_0\}$,
- $\ell_{BL} \in T_{BL} \rightarrow \Sigma_M \cup \{\tau\}$ with:

$$\ell_{BL} = \begin{cases} \ell(T), & \text{if } t \in T, \\ \tau, & \text{otherwise.} \end{cases}$$

LN_{BL} contains a set of added silent transitions, $\{t_{bl_M} | M \in MF\}$, consisting of one silent transition for each final marking $M \in MF$. $Backloop : MF \rightarrow T_{bl}$ is a bijective mapping from a final marking $M \in MF$ to a silent transition t_{bl_M}. A silent transition t_{bl_M} has all places in final marking M as input and place M_0 as output. The number of executions of backloop transitions $\{t_{bl_M} | M \in MF\}$ in the alignments of L on LN indicates the number of executions of traces of LN in L. Note that alignments require the model to be in a marking $M \in MF$ at the end of the alignment. This is impossible to obtain when pattern LN is absent in log L. Therefore, we set the final marking to $\{M_0\}$, allowing the alignments to make a complete sequence of *moves on log*, resulting in zero executions of the model.

Fig. 6(c) illustrates the concept of alignments through an example, showing the alignment of the non-fitting trace $\langle A, A, C, B, A, A, C, B, B, C \rangle$ on the model of Fig. 6(b). The top row of the alignments represents the behavior of the log, while the middle row and the bottom row represent the behavior of the model. $\gg$ indicates *no move*, with a $\gg$ in the top row indicating a *move on model* and in the middle row indicating a *move on log*. The model is able to mimic the first event of the trace by executing t_1 with label A, but is not able to mimic the second A in the log, resulting in a *move on log*. The C and B in the log can be mimicked (by t_3 and t_2 respectively). Next event A in the log can only be mimicked by the model by first firing t_{bl_1}, resulting in a *move on model*, represented by the $\gg$ in the log. Afterwards, A can be mimicked and another *move on log* is needed for the second A. C and B can again be mimicked, after which a *move on log* is again needed as the log cannot mimic t_{bl_1}. Would we not have prohibited *moves on models* on non-silent transition, the alignment could now have executed
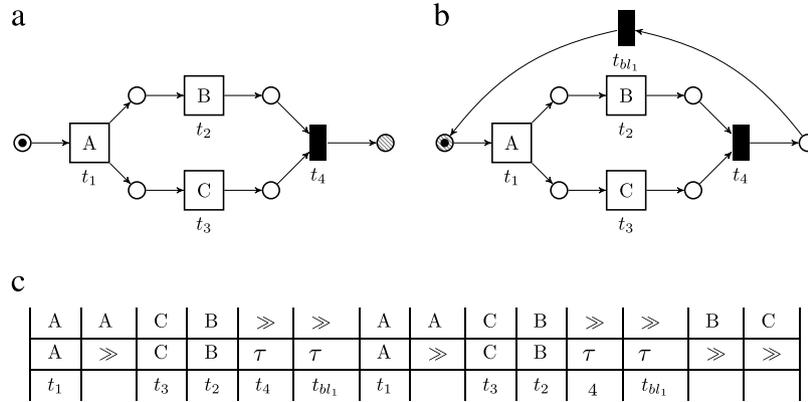
a

b

c

A	A	C	B	≫	≫	A	A	C	B	≫	≫	B	C
A	≫	C	B	τ	τ	A	≫	C	B	τ	τ	≫	≫
t_1		t_3	t_2	t_4	t_{bl_1}	t_1		t_3	t_2	4	t_{bl_1}		

Fig. 6 – (a) An example local process model, LN. (b) Process model LN_{BL}, constructed from LN by adding a silent connection from final to initial marking, and the final marking set to the initial marking. (c) Alignment of the trace t_1 on LN_{BL} when disallowing model moves on non-silent transitions.

Table 1 – Number of transitions enabled at each point in the alignment.									
A	C	B	τ	τ	A	C	B	τ	τ
t_1	t_3	t_2	t_4	t_{bl_1}	t_1	t_3	t_2	4	t_{bl_1}
1	2	1	1	1	1	2	1	1	1

a *move on model* on A, enabling synchronous moves on both B and C, falsely giving the impression that the LPM would have a third occurrence in the trace. As we prohibited the *model move* on A, the only option is to decide a *move on log* on B and C, thereby not counting the incomplete occurrence of the pattern.

LPM LN is evaluated on event log L by projecting L on the set of labels of LN, $L' = L{\upharpoonright}_{\Sigma_M}$. The middle row of the alignment of L' on LN_{BL} represents the segmentation λ^ξ_{LN}, where τ moves on a transition $t_{bli} \in \{t_{blm}|M \in MF\}$ indicates the start of a new segment. The alignment in Fig. 6(c) shows that $\lambda^\xi_{LN}(\langle A, A, C, B, A, A, C, B, B, C \rangle) = [\langle A, C, B \rangle^2]$.

6.1. Determinism on Petri nets

We now explain through an example how to calculate determinism for Petri nets. Each transition firing in a Petri net corresponds to a change in the marking of the net. Table 1 shows the transitions fired in the alignment of Fig. 6(c). The bottom row represents the number of transitions that were enabled in the Petri net when the transition fired. When t_3 fired, the Petri net was in a marking where both t_2 and t_3 were enabled. The determinism of the net corresponds to one divided by the average number of enabled transitions during replay. In the example, $determinism(LN, L) = \frac{10}{12}$.

7. Case studies

We now evaluate the proposed local process model mining method on two real life data sets.

7.1. BPIC'12 data set

The Business Process Intelligence Challenge (BPIC)'12 data set originates from a personal loan or overdraft application process in a global financial institution. We transformed the event log to obtain traces of all activities in a single day performed by one specific resource (bank employee). This resource was selected randomly to be resource id 10939. The event log for this specific resource contains 49 cases (working days), 2763 events, and 14 activities. Discovering the local process models with the approach described in this paper took 34 s on a machine with a 4-core 2.4 GHz processor using a support threshold of 0.7.

Fig. 7 shows the Petri net discovered for resource 10939 with the Inductive Miner infrequent with a noise threshold of 20%. The discovered model only contains 13 non-silent transitions, as the activity W_valideren aanvraag is filtered out by the Inductive Miner because of its low frequency. The process model in Fig. 7 is very close to a "flower model", which is the model that allows all behavior over its activities. The Inductive Miner without noise filtering returns exactly the flower model over the 14 activities in the log. The discovered process is unstructured because of a high degree of variance of the event log, which is caused by (1) the resource performing work on multiple applications interleaved, and (2) the resource only performing only a subset of the process steps for each application, and which process steps he performs might differ per application. For such a high-variance event log, it is likely that no start-to-end process model exists that accurately describes the behavior in the event log.

Fig. 8 shows five local process models discovered with the approach described in this paper, which give process insights that cannot be obtained from the start-to-end process model in Fig. 7. Local process model (a) shows that all occurrences of events of type O_SELECTED, O_CREATED, and O_SENT, occur in this exact order. Fig. 7 overgeneralizes by suggesting that for example O_SELECTED can be followed by three skip (black) transitions, after which another O_SELECTED or a A_ACCEPTED can be performed, which never happens in reality. O_SELECTED and O_CREATED in Fig. 7 can be separated by A_FINALIZED, which makes the dependency between
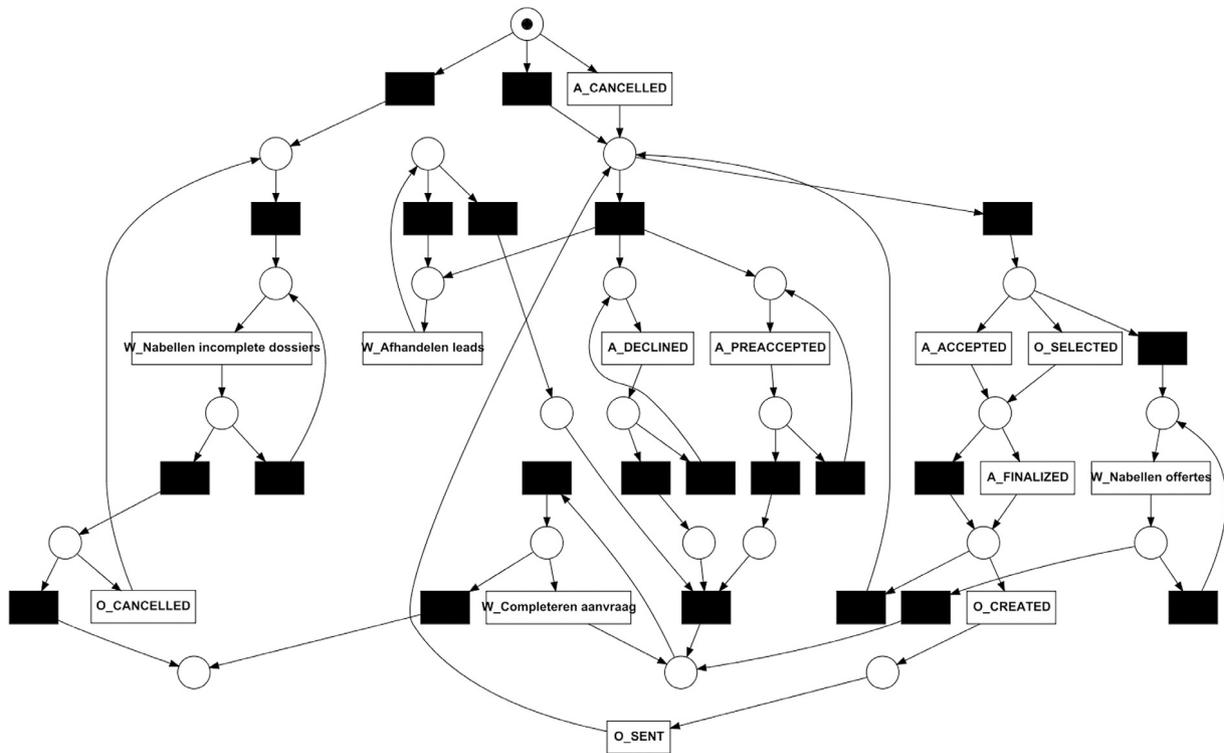
Fig. 7 – Process model of the behavior of resource 10939 in the BPIC'12 log, obtained using the Inductive Miner infrequent (20%).
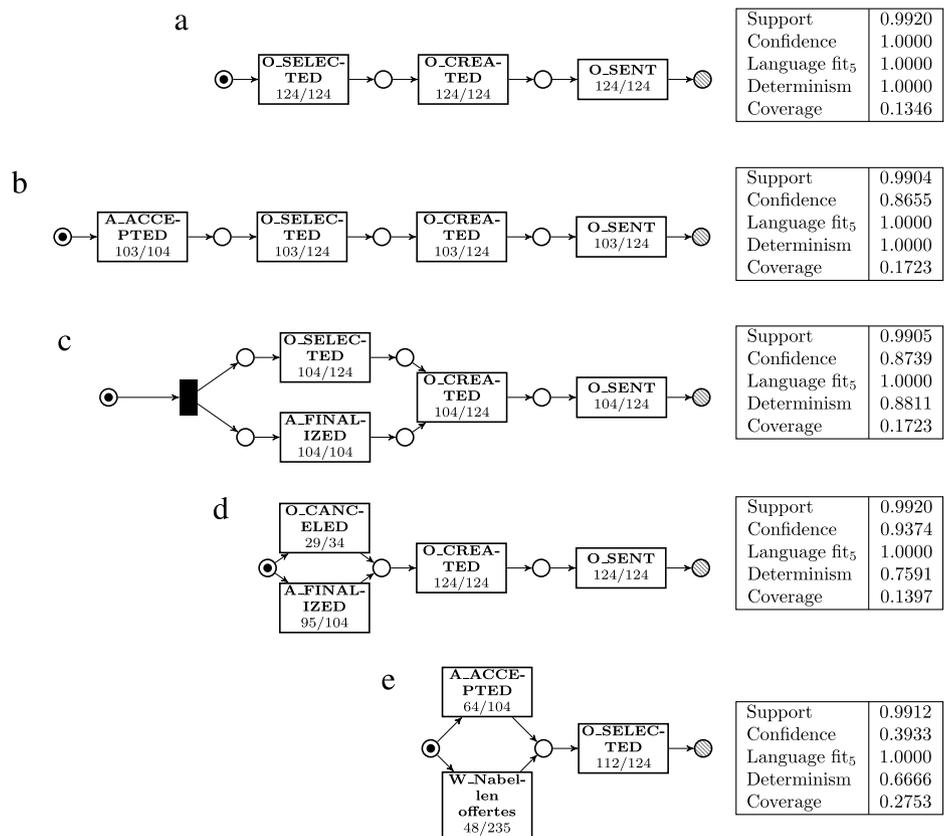

Fig. 8 – Five local process models discovered on the BPI'12 log using the technique presented in this paper. Clearly these models provide more insight than Fig. 7.

O_SELECTED and O_CREATED a long-term dependency, of which discovery is still one of the open problems in process mining [38] The local process model discovery method does find this long term dependency, because each local process model candidate is evaluated on a version of the event log that is projected on the set of labels of candidate under evaluation.

LPM (b) is an extension of LPM (a) as the last three activities in the sequence are the same, therefore, each occurrence of LPM (b) in the log will also be an occurrence of (a). LPM (b) starts with an additional activity A_ACCEPTED of which 103 out of 104 events follow this sequential pattern. The confidence of LPM (b) is lower than the confidence of (a), because only 103 out of 124 events of the last three activities of the sequence in LPM (b) can be explained by the model while each event of these activities is explained by LPM (a). From this we can conclude that there are 21 occurrences of the sequence O_SELECTED, O_CREATED, O_SENT that are not preceded by A_ACCEPTED. Partly this can be explained by A_ACCEPTED only occurring 104 times, however, the model also shows that there is one A_ACCEPTED event that is not followed by O_SELECTED, O_CREATED, and O_SENT. It might be the case that this A_ACCEPTED event does not fit the regular workflow, or alternatively it might be the case that the other process steps of after A_ACCEPTED were executed by a different resource. Note that the determinism of LPMs (a) and (b) is 1.0, since both LPMs are sequential. Language fit of both LPMs is also 1.0, since both allow for only one execution path, which is observed in the log.

Local process model (c) shows that all instances of A_FINALIZED are in parallel with O_SELECTED, and ultimately followed by O_CREATED and O_SENT. This is more informative than Fig. 7, which allows for much more behavior over activities A_FINALIZED, O_SELECTED, O_CREATED, and O_SENT.

Local process model (d) shows that each O_CREATED and O_SENT is preceded by either O_CANCELED (29 times) or A_FINALIZED (95 times). Also most of the O_CANCELED events (29 out of 34) and most of the A_FINALIZED events (95 out of 104) are followed by O_CREATED and O_SENT. Fig. 7 does not provide the insight that O_CANCELED is followed by O_CREATED and O_SENT. Note that the determinism of LPM (d) is lower than the determinism of LPM (c). This is in agreement with the intuition of determinism, as the concurrency at the start of LPM (c) can be regarded as a choice between two activities followed by a deterministic step of executing the other activity, while LPM (d) starts with a choice between two activities. After the concurrency in LPM (c) and the choice in LPM (d) respectively, the two models proceed identically. Local process model (d) has higher confidence than LPMs (b) and (c) as only five of the O_CANCELED and nine of the A_FINALIZED events cannot be explained by the model. LPM (d) has a higher confidence than LPM (c), mostly because all occurrences of O_CREATED and O_SENT could be aligned in LPM (d) while only 104 out of 124 could be aligned in LPM (c).

Notice that the number of events that were aligned on A_FINALIZED is lower in LPM (d) than in LPM (c). This indicates that there are six occurrences where the alignments aligned on O_CANCELED while it was possible as well to align on A_FINALIZED (as both occurred). Therefore, an inclusive choice construct would have been a more correct

representation than the exclusive choice that is currently included in the LPM. Note that our process tree based discovery approach allows for easy extension with additional operators, like e.g. an inclusive choice operator.

LPM (e) shows an example of a weaker local process model that performs lower on some quality metrics but can still be discovered with the described approach. The coverage of LPM (e) is much higher than the other models as W_Nabellen offertes (Dutch for "Calling after call for bids") is a frequently occurring event in the log. The confidence of LPM (e) is however much lower it explains only a fraction of the W_Nabellen offertes events.

7.2. *Comparison with related techniques*

In this section we apply some of the related techniques described in Section 3 to the event log of BPI'12 resource 10939 and compare the insights that can be obtained with those methods with the insights that we obtained with LPM discovery.

We start with the Declare miner [33], which mines a set of binary constraints from the data based on a set of constraint templates. Fig. 9(b) shows the result of the Declare miner [33] on the BPI'12 resource 10939 event log with a support threshold of 90%, requiring that the constraints hold in 90% of the cases. The model shows that a *choice* constraint holds between O_SELECTED and W_Nabellen offertes, indicating that on each working day either at least one event of type O_SELECTED or W_Nabellen offertes occurs. The same can be said about the pairs of event W_Nabellen offertes and O_SENT, W_Nabellen offertes and O_CREATED, and W_Nabellen offertes and O_Completeren aanvraag. Furthermore a *not chain succession* constraint is discovered between W_Nabellen offertes and O_SENT, indicating that W_Nabellen offertes and O_SENT never directly follow each other. *Not chain succession* constraints are also discovered between W_Nabellen offertes and O_SELECTED, and between W_Nabellen offertes and O_CREATED. Note that the none of the insights that we obtained from the LPMs in Fig. 8 could be obtained from this Declare model.

By lowering the support threshold parameter of the Declare miner to 80%, we obtain a larger set of constraints. An *exclusive choice* constraint is found between W_Valideren aanvraag and W_Nabellen offertes, indicating that 80% of the cases contain one of the two activities but not both. The same type of constraint is found between W_Valideren aanvraag and W_Completeren aanvraag. The rest of the constraints found are *not chain succession* constraints.

To find constraints that can be deduced from the LPMs of Fig. 8, such as the sequential ordering between O_SELECTED and O_CREATED from LPM (a), the support threshold would need to be lowered even further, leading to an overload of constraints being found by the Declare miner. Declare miner evaluates the constraints based on the ratio of cases in which the constraint holds. However, when activities are often repeated *within* cases, this is not a useful evaluation criterion. Employee 10939 performs most of the activities multiple times during a working day, therefore, to assess whether an activity *a* is generally followed by an activity *b* it is more useful to count the ratio of *occurrences* of activity *a* that
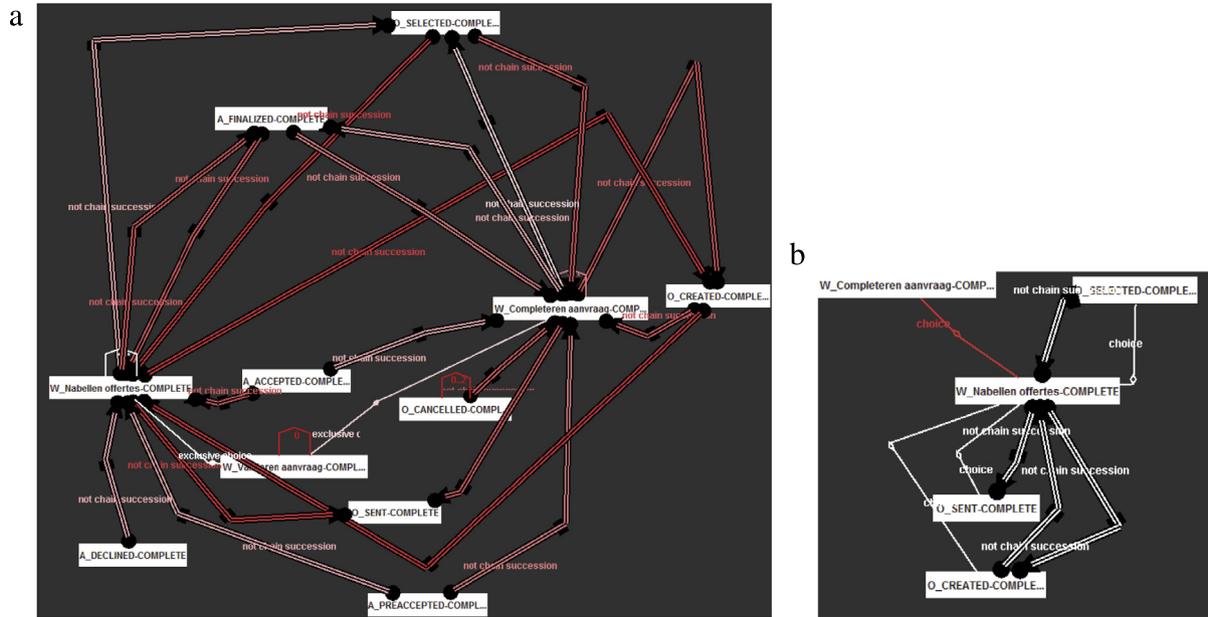
Fig. 9 – (a) The result of the Declare miner [33] with a support threshold of 80% on the BPI'12 resource 10939 event log. (b) The result of the same method on the same log with support threshold 90%.

are followed by *b* as in LPM discovery, instead of the number of cases that contain an *a* event that is followed by a *b* event.

Even more important is the fact that Declare miner is limited to binary constraints while LPM discovery mines *n*-ary relations. That is likely to be the cause of Declare mining not finding any of the LPM relations found in Fig. 8. At the same time this difference provides an explanation why Declare mining finds so many uninteresting *not chain succession* constraints between activities: when there are multiple *a* events in a trace, you are likely to find at least one *a* that is in a *not chain succession* relation with some activity *b*, leading to a high ratio of traces that fulfill such a *not chain succession* constraint.

Fig. 10 shows the result of the Fuzzy miner on the BPI'12 resource 10939 event log with default parameters. The discovered Fuzzy model does contain a path from *O_SELECTED* through *O_CREATED* to *O_SENT*, which were shown to be in a sequential relation by LPM (a). However, the Fuzzy model allows for many more paths, therefore the sequential relation between those three activities cannot be inferred from the Fuzzy model. LPM (c) showed a sequential path between *O_CREATED* and *O_SENT* that is preceded by an arbitrary ordering of activities *O_SELECTED* and *A_FINALIZED*. The Fuzzy model also shows arrows from both *O_SELECTED* and *A_FINALIZED* to *O_CREATED*, however, as a Fuzzy model does not make a distinction between parallelism, inclusive choice constructs and exclusive choice constructs, it does not answer the question whether *O_SELECTED* is preceded by *both* *O_SELECTED* and *A_FINALIZED*, or just by one of the two.

Fig. 11 shows the first four episodes found with ProM's Episode Miner on the BPI'12 resource 10939 event log. The first two episodes show the same sequential ordering from *O_SELECTED*, *O_CREATED*, and *O_SENT* that is represented by LPM (a). The first episode suggests that the *O_CREATED* event is optional, and can be skipped. LPM (a) however shows that

all of the *O_SELECTED* events are followed by an *O_CREATED* event, therefore it is never skipped. Episode (c) indicates that *O_SELECTED* and *O_CREATED* can happen in any order, but both of them have to occur before *O_SENT* occurs and episode (d) indicates that *O_SELECTED* has to happen before *O_SENT* and *O_CREATED* can occur. Episodes (a), (c) and (d) can be considered to be less specific versions of episode (b). ProM's Episode Miner is not able to discover patterns with choice constructs like LPM (d), or patterns with loops.

7.3. Gazelle data set

The *Gazelle* data set is a real life data set used in the KDD-CUP'2000 and contains customers' web click-stream data provided by the Blue Martini Software company. The Gazelle data set has been frequently used for evaluating sequential pattern mining algorithms. For each customer there is a series of page views, in which each page view is treated as an event. The data set contains 29 369 sequences (customers), 87 546 events (page views), and 1423 distinct event types (web pages). The average sequence length is three events. More detailed information on the Gazelle data set can be found in [39]. We compare the local process models found on this data set with the sequential patterns obtained with the well-known sequential pattern mining algorithm PrefixSpan [13] as implemented in the SPMF [14] sequential pattern mining library. We set the minimal support parameter of the sequential pattern mining algorithms to 10% of the number of input sequences. All obtained sequential patterns were also discovered by the local process model miner. Additionally, several non-sequential patterns were discovered that cannot be discovered with sequential pattern mining techniques, an example of which is shown in Fig. 12. This shows that this well-known sequential pattern mining evaluation data set contains frequent and high-confidence
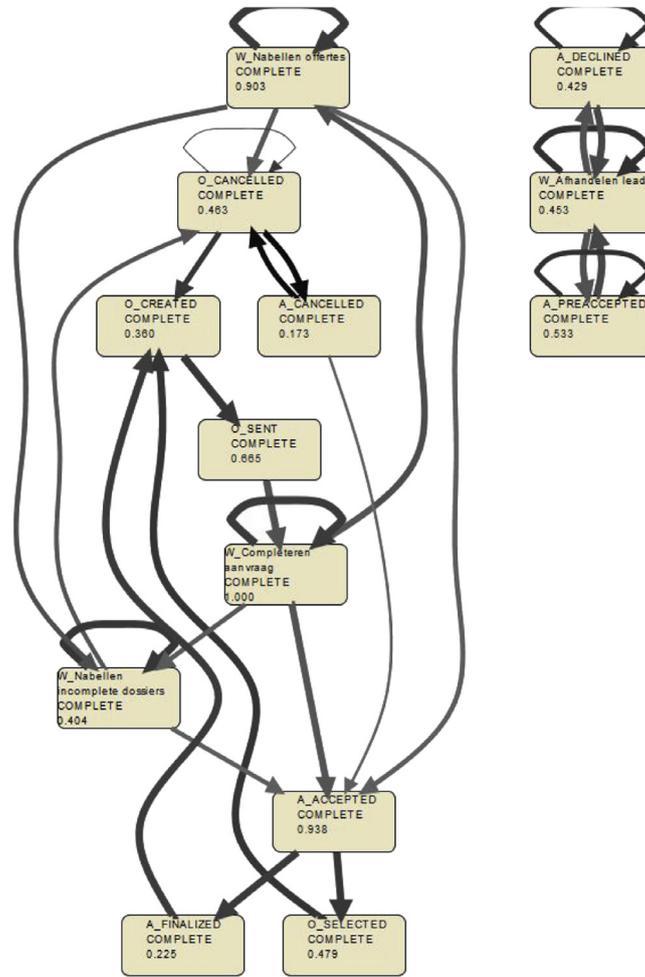
Fig. 10 – Result of the Fuzzy miner [35] with default parameters on the BPI'12 resource 10939 event log.
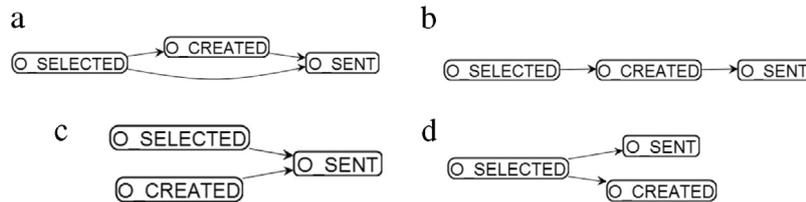

Fig. 11 – The first four episodes discovered by ProM's Episode Miner [18] on the BPI'12 resource 10939 event log.

patterns that cannot be found with sequential pattern mining approaches, but can be found with the local process model discovery approach. This indicates the applicability of local process model discovery to the field of pattern mining.

8.   Conclusion & future work

This paper presents a method to discover local process models that can express the same rich set of relations between activities as business process models, but describe frequent fragments instead of complete start-to-end processes. We presented five quality criteria and corresponding metrics
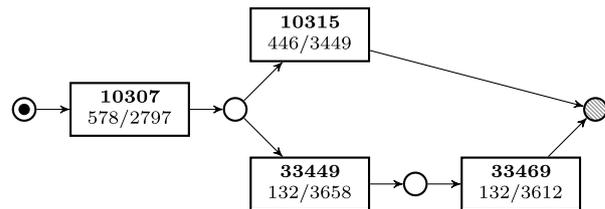

Fig. 12 – A non-sequential local process model discovered on the Gazelle data set.

quantifying the degree of representativeness of a local process model for an event log. We describe monotonicity prop-

erties of quality metrics that can be used to prune the search space and speed up computation. We illustrate through two case studies on real-life data sets that the proposed method enables the user to obtain process insight in the form of valuable patterns when the degree of randomness/variance of the event data prevents traditional process discovery techniques to discover a structured start-to-end process model. Furthermore, the proposed local process model discovery approach is able to discover long-term dependencies, which most process discovery approaches have difficulties with, as a result of evaluating the local process models on a projected version of the event log.

The computational time involved in discovering local process models rapidly grows with the number of activities in the event log. Therefore, we consider automatic discovery of projections on the event log (limiting search to a promising subset of the activities) to be an important area of future work, as it would enable the discovery of local process models on logs with larger numbers of activities. An alternative approach to deal with larger numbers of activities that is to be explored is the use of meta-heuristic search methods, e.g. simulated annealing, which allows partial exploration of the search space.

Finally, we consider it to be a relevant future direction of research to enhance local process models with guards, time information, and resource information.

REFERENCES

[1] W.M.P. van der Aalst, Process Mining: Data Science in Action, Springer, 2016.

[2] H. Mannila, H. Toivonen, A.I. Verkamo, Discovery of frequent episodes in event sequences, Data Min. Knowl. Discov. 1 (3) (1997) 259–289.

[3] R. Agrawal, R. Srikant, Mining sequential patterns, in: Proceedings of the 11th International Conference on Data Engineering, ICDE, IEEE, 1995, pp. 3–14.

[4] G. Casas-Garriga, Discovering unbounded episodes in sequential data, in: Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases, PKDD, Vol. 7, Springer Science & Business Media, 2003, p. 83.

[5] M. Atallah, W. Szpankowski, R. Gwadera, Detection of significant sets of episodes in event sequences, in: Proceedings of the 4th IEEE International Conference on Data Mining, ICDM, IEEE, 2004, pp. 3–10.

[6] W. Reisig, Petri Nets: An Introduction, Vol. 4, Springer Science & Business Media, 2012.

[7] Object Management Group, Notation (BPMN) version 2.0, OMG Specification.

[8] J.C.A.M. Buijs, Flexible evolutionary algorithms for mining structured process models (Ph.D. thesis), Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2014.

[9] L.R. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition, Proc. IEEE 77 (2) (1989) 257–286.

[10] C. Goller, A. Kuchler, Learning task-dependent distributed representations by backpropagation through structure, in: IEEE International Conference on Neural Networks, 1996., Vol. 1, IEEE, 1996, pp. 347–352.

[11] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.

[12] S.J.J. Leemans, D. Fahland, W.M.P. van der Aalst, Discovering block-structured process models from event logs - a constructive approach, in: Application and Theory of Petri Nets and Concurrency, Springer, 2013, pp. 311–329.

[13] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, M.-C. Hsu, PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth, in: Proceedings of the 17th International Conference on Data Engineering, ICDE, IEEE, 2001, pp. 215–224.

[14] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C.-W. Wu, V.S. Tseng, SPMF: a java open-source pattern mining library, J. Mach. Learn. Res. 15 (1) (2014) 3389–3393.

[15] J.C.A.M. Buijs, B.F. van Dongen, W.M.P. van der Aalst, A genetic algorithm for discovering process trees, in: Proceedings of the 2012 IEEE Congress on Evolutionary Computation, CEC, IEEE, 2012, pp. 1–8.

[16] G. Keller, A.-W. Scheer, M. Nüttgens, Semantische Prozeß-modellierung auf der Grundlage” Ereignisgesteuerter Prozeßketten (EPK)”, Institut für Wirtschaftsinformatik, 1992.

[17] International Organization for Standardization, ISO/IEC 19505-1:2012 - Information technology - Object Management Group Unified Modeling Language (OMG UML) - Part 1: Infrastructure.

[18] M. Leemans, W.M.P. van der Aalst, Discovery of frequent episodes in event logs, in: Data-Driven Process Discovery and Analysis, Springer, 2014, pp. 1–31.

[19] J. Lu, W. Chen, O. Adjei, M. Keech, Sequential patterns post-processing for structural relation patterns mining, in: Strategic Advancements in Utilizing Data Mining and Warehousing Technologies: New Concepts and Developments, 2009, p. 216.

[20] J. Lu, X.-F. Wang, O. Adjei, F. Hussain, Sequential patterns graph and its construction algorithm, Chinese J. Comput. 27 (6) (2004) 782–788.

[21] J. Lu, W. Chen, M. Keech, Graph-based modelling of concurrent sequential patterns, Exploring Adv. Interdiscip. Data Min. Anal.: New Trends (2011) 110.

[22] J.-Y. Jung, J. Bae, L. Liu, Hierarchical business process clustering, in: Proceedings of the 2008 IEEE International Conference on Services Computing, SCC, Vol. 2, IEEE, 2008, pp. 613–616.

[23] C. Diamantini, D. Potena, E. Storti, Mining usage patterns from a repository of scientific workflows, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC, ACM, 2012, pp. 152–157.

[24] C. Diamantini, L. Genga, D. Potena, E. Storti, Pattern discovery from innovation processes, in: Proceedings of the 2013 International Conference on Collaboration Technologies and Systems, CTS, IEEE, 2013, pp. 457–464.

[25] C. Diamantini, L. Genga, D. Potena, Behavioral process mining for unstructured processes, J. Intell. Inf. Syst. (2016) 1–28.

[26] B.F. Van Dongen, W.M.P. Van der Aalst, Multi-phase process mining: Building instance graphs, in: International Conference on Conceptual Modeling, Springer, 2004, pp. 362–376.

[27] M. Song, C.W. Günther, W.M.P. Van der Aalst, Trace clustering in process mining, in: Business Process Management Workshops, Springer, 2008, pp. 109–120.

[28] R.P. Jagadeesh Chandra Bose, W.M.P. van der Aalst, Context aware trace clustering: Towards improving process mining results, in: Proceedings of the 2009 SIAM International Conference on Data Mining, SIAM, 2009, pp. 401–412.

[29] F. Folino, G. Greco, A. Guzzo, L. Pontieri, Mining usage scenarios in business processes: Outlier-aware discovery and run-time prediction, Data Knowl. Eng. 70 (12) (2011) 1005–1029.

[30] B.F.A. Hompes, J.C.A.M. Buijs, W.M.P. van der Aalst, P.M. Dixit, J. Buurman, Discovering deviating cases and process variants using trace clustering, in: Proceedings of the 27th Benelux Conference on Artificial Intelligence, BNAIC, November, pp. 5–6.

[31] M. Pesic, H. Schonenberg, W.M.P. Van der Aalst, Declare: Full support for loosely-structured processes, in: Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International, IEEE, 2007, pp. 287–298.

[32] S. Schönig, C. Cabanillas, S. Jablonski, J. Mendling, Mining the organisational perspective in agile business processes, in: Enterprise, Business-Process and Information Systems Modeling, Springer, 2015, pp. 37–52.

[33] F.M. Maggi, A.J. Mooij, W.M.P. Van der Aalst, User-guided discovery of declarative process models, in: 2011 IEEE Symposium on Computational Intelligence and Data Mining, CIDM, IEEE, 2011, pp. 192–199.

[34] F.M. Maggi, T. Slaats, H.A. Reijers, The automated discovery of hybrid processes, in: Business Process Management, Springer, 2014, pp. 392–399.

[35] C.W. Günther, W.M.P. Van Der Aalst, Fuzzy mining–adaptive process simplification based on multi-perspective metrics, in: Business Process Management, Springer, 2007, pp. 328–343.

[36] R. Agrawal, R. Srikant, et al., Fast algorithms for mining association rules, in: Proceedings of the 20th International Conference on Very Large Data Bases, VLDB, Vol. 1215, Morgan Kaufmann, 1994, pp. 487–499.

[37] W.M.P. van der Aalst, A. Adriansyah, B.F. van Dongen, Replaying history on process models for conformance checking and performance analysis, Wiley Interdiscip. Rev.: Data Min. Knowl. Discov. 2 (2) (2012) 182–192.

[38] W.M.P. Van der Aalst, A.J.M.M. Weijters, Process mining: a research agenda, Comput. Ind. 53 (3) (2004) 231–244.

[39] R. Kohavi, C.E. Brodley, B. Frasca, L. Mason, Z. Zheng, KDD-cup 2000 organizers' report: Peeling the onion, ACM SIGKDD Explor. Newslett. 2 (2) (2000) 86–93.