Efficient Discovery of Relevant Maximal Behavioral Patterns from Event Logs

Mehdi Acheli[1] [0000−0001−9649−7127] (✉), Daniela Grigori[1], and Matthias Weidlich[2]

[1] Univ. Paris-Dauphine, CNRS UMR[7243], LAMSADE, 75016 Paris, France
mehdi.acheli@dauphine.fr
daniela.grigori@dauphine.fr
[2] Humboldt-Universität zu Berlin, Germany
matthias.weidlich@hu-berlin.de

Abstract. Techniques for process discovery support the analysis of information systems by constructing process models from event logs that are recorded during system execution. In recent years, various algorithms to discover end-to-end process models have been proposed. Yet, they do not cater for domains in which process execution is highly flexible, as the unstructuredness of the resulting models renders them meaningless. It has therefore been suggested to derive insights about flexible processes by mining behavioral patterns, i.e., models of frequently recurring episodes of a process' behavior. However, existing algorithms to mine such patterns suffer from imprecision and redundancy of the mined patterns and a comparatively high computational effort. In this work, we overcome these limitations with COBPAM, a combination-based algorithm for behavioral pattern mining. It exploits a partial order on potential patterns to discover only those that are maximal, i.e., most precise and relevant, i.e. most informative. Moreover, COBPAM exploits that complex patterns can be characterized as combinations of simpler patterns, which enables pruning of the pattern search space. Efficiency is improved further by evaluating potential patterns solely on parts of an event log. A case study with real-world data demonstrates how COBPAM improves over the state-of-the-art in behavioral pattern mining.

Keywords: Behavioral Patterns · Process Discovery · Pattern Mining.

1 Introduction

The research area of process mining connects the disciplines of data mining and machine learning with process modeling and analysis [1]. Specifically, the analysis of information systems may be supported by exploiting the event logs recorded during their execution. Techniques for process discovery use such event logs as a starting point and construct a model of the underlying end-to-end process. Recently, a plethora of process discovery algorithms has been proposed [3]. These algorithms impose varying assumptions on the event log used as input, e.g., in terms of the event model [21]; adopt different target languages, e.g., Petri-nets [22],

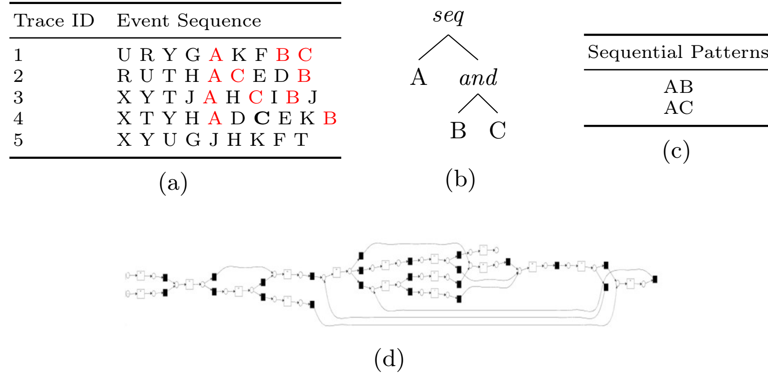

Fig. 1: (a) Event log; (b) behavioral pattern for the log; (c) 2-length sequential patterns extracted with PrefixSPAN [18]; (d) end-to-end model mined by FHM [27].

process trees [15], or BPMN [8]; and differ in how they cope with noise and incompleteness, e.g., avoiding over-fitting [28] or filtering noise [6].

Most existing discovery algorithms, however, aim to unify all the behavior observed in the log into an end-to-end model. As such, they are not suited for domains in which process execution is highly flexible, as the resulting models are unstructured and are subject to over-generalization [1]. The reason being that there is a large variability of the behavior of different process instances and a model capturing all variations tends to be too complex. Against this background, it was suggested to derive insights about flexible processes by mining behavioral patterns [25, 26]. These patterns are formalized as process models, yet they capture only comparatively small episodes of a process' behavior that occur frequently. The basic idea behind behavioral patterns is illustrated in Fig. 1. For the example log, given as a set of traces, i.e., sequences of events that denote the executions of different activities, a traditional discovery algorithm such as the Flexible Heuristics Miner (FHM) [27] would yield a very complex model. However, one may observe that the traces show a specific behavioral pattern: An execution of activity A is followed by B and C in parallel. Detecting such a pattern provides a general understanding of the regularities in process execution. Note though, that such a pattern cannot be detected using standard techniques for sequential pattern mining, such as PrefixSPAN [18], as those would miss complex behavioral dependencies such as concurrency and exclusive choices.

However, existing algorithms [25, 26] to mine behavioral patterns suffer from imprecision of the mined patterns, redundancy in the output and a comparatively high computational effort. Here, imprecision means that mined patterns are not maximal, i.e., while certain behavior is frequent, patterns that capture only a part of some frequent behavior are discovered. For instance, in Fig. 1, the pattern *seq(a,b)* is frequent. Arguably, this pattern does not lead to new insights on the process if the pattern *seq(a, and(b,c))* is known, so that it is sufficient to discover the latter one. At the same time, existing mining algorithms suffer from high

run-times that are due to the fact that pattern candidates are always evaluated based on the complete event log.

In this paper, we overcome the above limitations with COBPAM, a novel combination-based algorithm to mine behavioral patterns that are formalized as process trees. It identifies all trees of which the behavior can be found in a certain number of traces of the event log, which takes up the notion of support proposed for patterns in sequence databases [10]. Moreover, we consider a notion of language fitness to assess how strongly a tree materializes. Based thereon, the contributions of COBPAM are threefold:

(1) It defines a partial order on potential patterns to discover only those that are maximal and relevant, thereby improving the effectiveness of behavioral pattern mining.
(2) It efficiently explores the pattern search space by pruning strategies, exploiting that complex patterns are combinations of simpler patterns.
(3) It further improves efficiency by considering only a subset of traces, when evaluating the support and language fitness of a potential pattern.

The paper is structured as follows. Section 2 reviews related work on process discovery and mining of sequential and behavioral patterns. Preliminaries are given in Section 3. We then define algebraic operations and structures on potential behavioral patterns in Section 4, while quality metrics for them are presented in Section 5. Our novel mining algorithm for behavioral patterns, COBPAM, is introduced in Section 6. We evaluate the algorithm in a case study with real-world event logs in Section 7, before we conclude and discuss future research directions in Section 8.

2 Related Work

The discovery of behavioral patterns defined with respect to their frequency in a log connects two research areas: sequential pattern mining and process mining. In this section, we will mention the algorithms in the former area that inspired our work and then proceed with an overview of process mining algorithms that aim to derive insights for event logs that have been recorded for flexible processes.

GSP [24] is a sequential pattern mining algorithm that combines pairs of sequential patterns of length k to obtain patterns of length $k+1$. As will be discussed later, we adopt this principle for COBPAM when generating behavioral patterns. We also borrow the concept of a projected database in the form of log projections from the PrefixSPAN algorithm [18] to evaluate the pattern candidates on the minimal number of traces possible. Moreover, we adopt the maximality principle as discussed for sequential patterns in [10].

Trace clustering is an active research area concerned with inferring insights from logs of flexible processes [4, 5, 11, 23]. These techniques group traces into homogeneous clusters such that process discovery techniques applied on each cluster yield comparatively structured models. Such techniques are well-suited if the absence of structure appears in the disparity between traces. Yet, they do

not cater for cases, where the flexibility in process execution is seen inside single traces.

The Fuzzy miner [12] targets the domain of flexible processes and generates simplified and abstracted process models describing only the most significant behavior. It can be customized through different metrics and parameters to control the level of aggregation and abstraction of events and relationships. Nevertheless, it fails to mine certain behavioral structures, such as concurrence and choices.

The Declare Miner [16] discovers a set of rules that are satisfied by a certain share of traces. These rules come in the form of relationships between activities, such as two activities being always executed together in a trace, potentially in a fixed order. These rules relate to the presence or the absence of behavior. Each rule, however, is limited to a relationship between at most two activities, while our approach considers patterns with an arbitrary number of activities.

The Episode Miner [14] is another algorithm that discovers frequent patterns with partial order constructs. It consists of sets of activities linked by follows relationships. The method, however, does not support loops and choices constructs.

Our work is inspired by the discovery of Local Process Models (LPMs) [26]. Specifically, we also adopt the model of process trees to represent behavioral patterns that are observed in event logs recorded in unstructured domains. However, we note that the existing algorithms for LPM discovery limit the size of the patterns and are not grounded in the traditional definition of support and traces, as known from sequence databases. Rather, when mining LPMs, a trace does not only represent an instance of a specified process, but potentially encompasses several executions. Moreover, the algorithm of [26] follows a generate-and-test approach, where only frequent trees are expanded by replacing an activity with a small behavior. This way, the same tree can be evaluated multiple times and infrequent trees that may become frequent when joined by a choice operator are completely discarded. In addition, choice involving trees generated following the method are considered irrelevant in our algorithm since any frequent tree added to another behavior is frequent. On the other hand, the method performs alignments over the entire log which reveals to be computationally heavy. Compared to the mining of LPMs, COBPAM adopts a well-established definition of support and introduces number of improvements such us advanced pruning, enhanced output and targeted evaluations on subsets of the log. Note that the initial approach to discover LPMs [26] has been extended to include goal-driven strategies to mine patterns based on their utility and constraints satisfaction [25]. Yet, these extensions are orthogonal to our work.

3 Preliminaries

This section presents basic definitions. We begin with the notion of event log.

Definition 1. *Considering a set of events (activities) A, an event log L is a multiset of traces where each trace is a finite sequence of events $e \in A$. A^* denotes*

the set of all sequences over A and a trace $\sigma \in A^*$ *can be noted* $\langle e_1 e_2 ... e_n \rangle$. $|L|$ *denotes the size of the event log, meaning the number of traces it contains.*

Process discovery techniques take as input event logs and output process models in a certain representational language. One of these languages is process trees [7]. They have the particularity of modeling only sound process models and can be converted to many other notations : Petri nets [19], BPMN [17], EPC [20], UML activity diagrams [13] or UML statechart diagrams [13].

Definition 2. *A process tree is an ordered tree structure such that leaf nodes represent activities and non-leaf nodes represent operators. Considering a set of activities A, a set of operators* $\Omega = \{seq, and, loop, xor\}$*, a process tree is recursively defined as follows:*

- $a \in A$ *is a process tree.*
- *considering an operator* $x \in \Omega$ *and two process trees* P_1, P_2, $x(P_1, P_2)$ *is a process tree having x as root,* P_1 *as left child and* P_2 *as right child.*

The language of a process tree $\Sigma(P)$ *is the set of traces it generates. In other words, it's the behavior it models. The semantics of each operator are presented below :*

- $seq(P_1, P_2)$ *whose behavior is the behavior of* P_1 *followed by the behavior of* P_2*. For example, considering two activities* $a, b \in A$, $\Sigma(seq(a, b)) = \{\langle a, b \rangle\}$.
- $and(P_1, P_2)$ *whose behavior is the behavior of* P_1 *in parallel with the behavior of* P_2*. This operator is symmetrical; meaning :* $\Sigma(and(a, b)) = \Sigma(and(b, a)) = \{\langle a, b \rangle, \langle b, a \rangle\}$.
- $xor(P_1, P_2)$ *whose behavior is either the behavior of* P_1 *or that of* P_2*. This operator is also symmetrical.* $\Sigma(xor(a, b)) = \Sigma(xor(b, a)) = \{\langle a \rangle, \langle b \rangle\}$.
- $loop(P_1, P_2)$ *is applied to two children processes. A do part* P_1 *and a redo part* P_2*. The behavior of the process starts by* P_1 *and then loops back and forth from* P_2 *to* P_1. $\Sigma(loop(a, b)) = \{\langle a \rangle, \langle aba \rangle, \langle ababa \rangle ...\}$*. It should be noted that the do part "a" is part of the language even without the looping over the redo part "b". On another hand, since the language associated to a loop operator is infinite, we define the n-language of a tree* $\Sigma_n(P)$ *as the set of language traces where each loop is traversed n times, e.g.,* $\Sigma_1(loop(a, b)) = \{\langle a \rangle, \langle aba \rangle\}$

4 Algebraic Operations and Structures on Process Trees

Now that we defined our process modeling language, we devise a method for constructing process trees incrementally. We propose to combine two process trees composed of n activities to get process trees of $n + 1$ activities. The process trees combined must be identical in every way except at the level of one leaf. We impose conditions on these leaves as follows:

Definition 3. *Given a process tree P of depth i, a leaf a of depth d is called potential combination leaf if and only if* $d \geq i - 1$ *and there is no leaf b of depth* d' *on the left of a such that* $d' > d$.

Two process trees that can be combined are called seeds.

Definition 4. *P_1 and P_2 are called seeds if they contain two potential combination leaves (a in P_1 and a' in P_2) such that by replacing a in P_1 with a', we obtain P_2. In other words, they are identical in every way except at the level of the leaves a and a'. For instance, seq(a, b) and seq(a, c) are seeds.*

The algebraic operation of combination resulting in new process trees is defined formally as:

Definition 5. *A combination of two seeds P_1 and P_2 through an operator x is an operation generating two process trees. Starting from P_1, the combination leaf a is replaced by the operator x whose children are set to a and a'. a is left child in one resulting tree and right child in the other. a and a' are called the combination leaves and x is called the combination operator. The Fig. 2a shows an example of a combination of two process trees seq(a, b) and seq(a, c) through the concurrence operator resulting in the two trees : seq(a, and(b,c)) and seq(a, and(c,b)).*

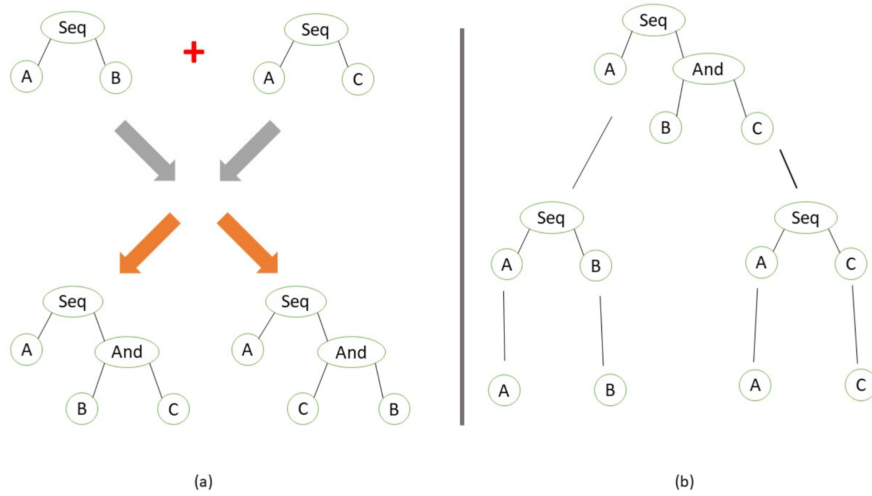

Fig. 2: (a) a combination operation of two process trees (b) the construction tree of the process tree *seq(a, and(b,c))*

Thanks to the conditions characterizing the potential combination leaves, the following theorem holds:

Theorem 1. *Considering a process tree P of depth $i \geq 1$, there is a unique pair of seeds P_1 and P_2 whose combination through a certain operator x results in P. P_1 and P_2 are called "the" seeds of P and x is called the defining operator of P.*

Proof. Let's consider a process tree P of depth i. It will contain at least two leaves of depth i since all operators have two children. Let $2n$ ($n \geq 1$) be the number of leaves of depth i in P. We will consider two cases:

$n = 1$**:** P has only two leaves a_j and a_k of depth i that are children nodes of some operator x. In parallel, let us consider two process trees P_1 and P_2 which are the same as P except that the node x is replaced with the activity a_j in P_1 and with a_k in P_2 leaving thus P_1, P_2, a_j and a_k with depth $i-1$. Now, since there are no deeper leaves on the left of a_j (resp. a_k) in P_1 (resp. P_2), P can be obtained by applying the combination operation on P_1 and P_2 through the operator x.

Moreover, there is only a unique couple of seeds whose combination results in P. To prove that, we suppose that there exists another couple P'_1, P'_2 that can produce P when combined. This means that P'_1, P'_2 are similar except for a single leaf (a'_j in P'_1 and a'_k in P'_2). The two leaves appear in P under a certain operator x' which corresponds to the combination operator. Besides, considering how the combination is realized, any other branch in P appears both in P'_1 and P'_2 which means the depth of these two processes is i since the branches in P leading to a_j and a_k are of depth i.

However, the combination being performed means that, in P'_1 and P'_2, a'_j and a'_k are of at least depth $i-1$. This leaves two cases:

- a'_j and a'_k are of depth $i-1$ in P'_1 and P'_2 meaning their depth in P after the combination is i which contradicts the hypothesis that there exists only two leaves a_j and a_k of depth i in P.
- a'_j and a'_k are of depth i in P'_1 and P'_2 meaning their depth in P after the combination is $i+1$ which contradicts the hypothesis that P is of depth i.

We conclude that P'_1 and P'_2 cannot exist and that P_1 and P_2 are the unique seeds that produce P.

$n > 1$**:** In this case, P has at least four leaves of depth i. Let a_j be the leftmost deepest leaf, x its parent operator and a_k the other child of x. We then consider two process trees P_1 and P_2 that are the same as P except that x is replaced with a_j in P_1 and with a_k in P_2. These process models are of depth i and contain $2(n-1)$ leaves of the same depth. Moreover, by construction, there are no deeper leaves on the left of a_j (resp. a_k) in P_1 (resp. P_2) and a_j (resp. a_k) is of depth $i-1$ in P_1 (resp. P_2). That means P_1 and P_2 can be combined resulting in P.

In addition, these two processes are the unique pair whose combination outputs P. Again, to prove this, we will consider another pair P'_1 and P'_2 which, when combined results in P. This means, the only difference between them is a single leaf (a'_j in P'_1 and a'_k in P_2). These two leaves appear in P under a certain operator x' which corresponds to the combination operator between P'_1 and P'_2. Besides, any branch leading to a leaf different than a'_j and a'_k present in P also appears in P'_1 and P'_2. Hence, the depth of the two fragments is also i and the leaves a'_j and a'_k are of at least depth $i-1$ in P'_1 and P'_2. We will separate the cases :

- a'_j and a'_k are of depth $i-1$ in P'_1 and P'_2 meaning their depth in P is i and since a_j is the leftmost deeper leaf, then it's on the left of a'_j and a'_k in P.

Thus, there exists a deeper leaf (of depth i) on the left of a'_j (resp. a'_k) in P'_1 (resp P'_2) which means P'_1 and P'_2 are not seeds and that contradicts the combination conditions.
- a'_j and a'_k are of depth i in P'_1 and P'_2 meaning their depth in P is $i+1$ which contradicts the hypothesis that P is of depth i.

We conclude that P'_1 and P'_2 can't exist and that P_1 and P_2 are the unique couple of process trees that produce P. In conclusion, any process tree of depth $i \geq 1$ is the result of a combination of two unique seeds. □

Since every process tree of depth superior to zero results from the combination of two unique seeds, we can introduce an additional structure.

Definition 6. *Given a certain process tree P of depth $i \geq 1$, we define its unique construction tree whose nodes are process trees. The root is P, the leaves are trees with single activity nodes and each node has for children its seeds. The leaves are in fact, the activities that appear in P. The Fig. 2b shows the construction tree of the process tree seq(a, and(b,c)).*

Moreover, a more complex graph structure can be defined.

Definition 7. *Let A be a set of activities. We define the construction graph over A. An infinite oriented acyclic graph whose nodes are all possible process trees. An edge links a node n_1 to another n_2 if and only if n_1 is seed of n_2. We say that n_2 contains n_1 through the defining operator of n_2.*

To identify each tree, COBPAM uses the concept of representative word.

Definition 8. *We assign to each process tree P a unique identifier $RW(P)$. It's a sequence of characters called representative word. We construct it through the pre-order traversal of its nodes while outputting the name of activities and operators. For example, the representative word of $P = seq(a, and(b, c))$ is "(a (b c and) seq)".*

5 Quality Metrics

In this section, we formally define the quality metrics. In order to calculate them, we call out for the conformance checking side of process mining. Basically, we need a boolean function $\epsilon(\sigma, P)$ that returns true if the trace σ fits the process tree P, false otherwise. We use alignments [2] that try to align a process tree i.e. the modeled behavior with the log i.e. the observed behavior. For that matter, we convert the tree P to a Petri net and try to replay the trace on it. We set two tokens, one at the start position of the Petri net and another at the start of the trace. The goal is to move synchronously on the log and on the model. This is not always possible. If a behavior is recorded on the log but shouldn't happen according to the process model, a move on log is stored in the alignment and the token ignores the recorded action. On the contrary, if a behavior is modeled and should appear in the log at a certain place but doesn't, we register a move on

model and the token ignores the modeled behavior. Each of these operations are unwanted as they represent deviations between the log and the Petri net under evaluation. The goal is thus to find an optimal alignment that minimizes the number of these operations while advancing the tokens to the end positions of both the Petri net and the trace. In fact, as in [26], we only allow moves on log. If the other type of moves is applied too, a trace would be said to fit the Petri net and by extension the process tree while it fits only a part of its behavior. Besides, thanks to the alignments, the exact behavior exhibited by the trace among all the behaviors allowed by the model is identified. We suppose it's the result of a function $\upsilon(\sigma, P)$. For example, in Fig. 1, the behavior exhibited by the trace (1) is $\langle ABC \rangle$ while the one exhibited by trace (2) is $\langle ACB \rangle$. Both behaviors are part of the language of the pattern discovered.

We employ these functions to define the concept of projection and the quality metrics manipulated by COBPAM.

Definition 9. *A projection is a subset of a log associated to a process tree P that contains the traces it can be aligned with. In other words, it's the set of traces that exhibit its behavior.*

$$proj(P) = \{\sigma \in L | \epsilon(\sigma, P) = 1\}$$

Definition 10. *Given a log L, the frequency of a process tree P is the number of traces that exhibit its behavior. Meaning, it's the size of its projection:*

$$frequency(P) = \sum_{\sigma \in L} \epsilon(\sigma, P) = |proj(P)|$$

Its support, on the other hand, is the frequency over the size of the log:

$$support(P) = \frac{frequency(P)}{|L|}$$

Definition 11. *Given a log L, the language fitness of a process tree P is the ratio between the behavior seen in the log and all the behavior allowed by the model :*

$$language_fitness(P) = \frac{|\{\upsilon(\sigma, P) | \sigma \in L \wedge \epsilon(\sigma, P) = 1\}|}{|\Sigma(P)|}$$

If P contains loop operators, its language will be infinite and so its language fitness will tend to zero. In this case, we use the n-language of P:

$$language_fitness(P) = \frac{|\{\upsilon(\sigma, P) | \sigma \in L \wedge \epsilon(\sigma, P) = 1\}|}{|\Sigma_n(P)|}$$

6 COBPAM, Mining Behavioral Patterns through Combinations

In this section, we present a new algorithm to mine process trees that depict frequent behaviors observed in a log. The main idea is to explore the construction

graph starting from single activities. Each process tree is evaluated against a portion of the log that may exhibit its behavior in order to calculate its quality metrics. We also use what we call projection and pruning rules to limit the number of evaluated process trees and the number of traces they are tested upon. Furthermore, we only output relevant maximal process trees.

In *Section 6.1*, we bring attention to the notion of equivalence between process trees. We then state a monotonocity property characterizing the search in *Section 6.2*. We continue with the definition of what we consider relevant and maximal process trees in *Section 6.3* and introduce next the projection rules in *Section 6.4*. In *Section 6.5*, a detailed view of the algorithm is given.

6.1 Equivalence between Process Trees

Since potential patterns are represented as trees, we define two types of equivalence:

- **Behavioral equivalence:** two trees are behaviorally equivalent if their languages are equal. In other words, the set of traces they generate are the same. For example, *seq(seq(a,b),c)* is behaviorally equivalent to *seq(a,seq(b,c))*.
- **Syntactical equivalence:** This equivalence appears as a direct result to the existence of symmetrical operators in the process trees. Due to the interchangeability of the children of such operators, many similar versions of a process tree can exist. Ex: *seq(a, and(b,c))* and *seq(a, and(c,b))*. In fact, syntactically equivalent trees are also behaviorally equivalent.

Spotting behaviorally equivalent process trees reveals to be computationally heavy since some of them have infinite languages. However, outputting syntactically equivalent trees may be avoided by forcing an order on the children of each symmetrical operator. We adopt the lexicographical order of the leaves so that the left child must be inferior to the right child. In the case where either or both children are sub-trees, the comparison is performed according to what we call representative leaves. The representative leaf of a sub-tree is the lowest leaf in the lexicographical order among all the leaves it contains. As a result, the form *seq(a, and(c,b))* is not valid and we would output only *seq(a, and(b,c))*.

6.2 Monotonicity Property

The combination operation replaces a potential combination leaf with a subtree representing a portion of behavior that either extends the tree behavior when using the choice operator or constraints it when using a sequence, loop or concurrence operator. When evaluating a tree whose defining operator is a sequence, a loop or a concurrence operator, we essentially want that the trace exhibits all the behavior of its seeds except at the position of the combination leaf. There we want an additional behavior replacing the appearance of an activity in the trace. The shared behavior between a process tree and its seeds represents a context to which the additional behavior is joined. In consequence, if a trace

doesn't exhibit the context, there is no need to evaluate the added behavior. We can thus infer that if one of the seeds isn't frequent, meaning there isn't enough traces that exhibit the context, there is no need to evaluate the tree which will not be frequent too. This is a monotonicity property on the support metric. If considering two seeds, one isn't frequent then any process tree resulting from their combination through the constraining operators (sequence, loop and concurrence) isn't frequent. In other words, the seeds of a frequent process tree obtained through the constraining operators are both frequent. Thanks to this property, we can specify a first pruning rule. If a seed is infrequent, it won't be used in a combination operation using the sequence, loop or concurrence operator.

6.3 Relevant and Maximal Process Trees

Relevant process trees are ones that bring insight to the analyst. In our method, we discard three types of irrelevant process trees:

- the ones that have the choice operator for root. Indeed, a choice at that level makes the behavior of the model a union of completely separate behaviors. However, the union of a certain number of behaviors is bound to add up and make the model frequent. Thus, a model constructed this way is of little interest and is comparable to a trace model.
- the ones resulting from a combination through a choice operator and one of the seeds is frequent. Actually, if a tree is frequent, combining it with any other tree through the choice operator results in a frequent process tree. Hence, we can define a second pruning rule : when performing a combination using the choice operator, both seeds must be infrequent.
- the ones displaying a loop operator applied to two children: $loop(P_1, P_2)$ such that only the do part P_1 appears in the log. Indeed, since the language of the loop operator allows for the do part to appear without looping over the redo part, the appearance of the behavior of P_1 is an instance of the behavior of $loop(P_1, P_2)$ and an instance of $loop(loop(P_1, P_2), P_3)$ and so on. To be relevant, a loop behavior is not validated until the redo part appears at least once.

On another hand, since the monotonicity property states that a frequent process tree P whose defining operator is a constraining operator has both its seeds as frequent, we can output only P as its seeds can be simply derived and are known to be frequent. In other words, we could say that P is representative of its seeds. Furthermore, by transitivity, P is representative of the paths in the construction tree composed solely of trees defined by constraining operators. More generally, in the context where we search for frequent behavioral patterns of at most depth i, we output frequent process trees that are not contained through a constraining operator in another frequent process tree of depth inferior to i. Those are maximal behavioral patterns. In the example of Fig. 1, the trees *seq(a,b)* and *seq(a,c)* are frequent but not maximal since they are contained in

seq(a,and(b,c)) as shown through the construction tree in Fig. 2b. Behaviorally, maximal process trees represent the most complex behavior that includes different behaviors of lesser complexity. They are the most interesting to the analyst since they give a comprehensive accurate view.

6.4 Projection Rules

We recall that the goal of the method is to retrieve frequent process trees. Its speed depends on the size of the construction graph which increases in an exponential manner when the number of activities increases and on the size of the log on which the trees are evaluated. A few rules can help accelerate the patterns extraction by limiting the evaluation of the trees to portions of the log. The following rules involving projections are used to assess the frequency of a process tree against the minimum number of traces :

- When performing a combination through a constraining operator, the behavior associated with the resulting process trees may only appear in the intersection of the projections of the seeds. As a result, we only consider the said intersection in order to calculate the quality metrics. Moreover, the size of the intersection of the seeds projections represents an upper bound for the frequency of the resulting trees. Therefore, if the upper bound is less than the frequency threshold, the combination is canceled. This represents a third pruning rule.
- When performing a combination using the choice operator, the projection associated with the resulting process trees is the union of the projections of the seeds (the idea is that either one of the combination leaves has to be in a trace since it's a choice based decision). Moreover, the frequency of the resulting trees can be precisely derived and corresponds to the size of the union of the seeds projections. On another hand, the language seen of the new trees is the union of the languages seen of the seeds and their language is the union of the languages of the seeds. This allows us to calculate the language fitness without resorting to the alignments.

6.5 COBPAM Algorithm

COBPAM is an algorithm that prioritizes computation speed. As such, we employ a heuristic that further prunes the construction graph by limiting the search to a portion more dense in terms of frequent trees. Indeed, instead of generating the process trees starting from all the activities, we construct a restricted set Δ. In it, we only consider frequent activities. We also consider frequent combinations of two infrequent activities through the choice operator. Each combination will be named and considered as a single activity in the rest of the algorithm.

In fact, by providing frequent activities as the initial seeds of the generation, we insure a higher probability of constructing frequent process trees or more precisely, we direct the search to a region dense with behavioral patterns. Since we take into consideration a majority of three constraining operators out of

four, involving infrequent activities will render infrequent any early combination through these operators. By eliminating them, we prune a major portion of infrequent trees in the search space.

However, any pruning has a drawback. Due to the existence of the choice operator, once infrequent trees can be combined to reintroduce frequent ones. So eliminating trees from the search on the basis that they are infrequent will eliminate any frequent tree that would be constructed out of them. In conclusion, pruning makes the method miss some behavioral patterns involving the choice operator. We keep it however because without it, we would be forced to evaluate the support of trees that we know are infrequent and this evaluation is computationally heavy since it involves calculating alignments. Besides, three out of four operators benefit from the pruning and that makes it a majority. Indeed, the number of missed frequent process trees is small compared to the number of infrequent pruned trees that make the method gain in computation time.

Concerning the algorithm, we build incrementally sets of process trees. In order to respect the pruning rules, we consider sets containing only frequent trees and others only infrequent ones. The former type will serve to perform combinations through the constraining operators while the latter will serve for choice based combinations. Inside the same set, all trees are identical except at the level of one leaf. In fact, any two trees are seeds. In other words, they can be combined. Moreover, since the difference between two seeds is a unique leaf, we associate to each set an identifier in the form of a representative word. It's a mask that applies to any of the representative words of the process trees it contains. Take for example the tree *seq(a,b)*. Its representative word is *(a b seq)*. This process tree can be added to the set defined by *(a _ seq)*. The underscore indicates that any activity can be inserted at that position. In other words, any other tree, take *seq(a,c)*, can be added to the set by replacing the underscore with an activity; here, c. The underscore is always at the position of a potential combination leaf. This way, any two trees inside the set can be combined.

The algorithm revolves around three functions: $evaluate(P, sublog)$ which calculates the quality metrics of P over the portion of the log supposed to contain its behavior $sublog$, $addFreq(P, \Gamma)$ that adds the process tree P to a set Γ containing only frequent trees and $addInfreq(P, \gamma)$ that adds P to a set γ containing only infrequent ones. We denote Θ the set of frequent relevant maximal trees. It represents the set to be outputted, so the trees in it must satisfy a certain language fitness threshold. Moreover, since it contains only maximal trees, each time a frequent tree defined by a constraining operator is added to it, part of its construction tree is deleted. Let's start by introducing $evaluate(P, sublog)$:

1. use alignments to calculate the quality metrics of P.
2. save the traces where the behavior of P appears as $proj(P)$.

We also define $addFreq(P, \Gamma)$ as :

1. for each process tree P' in Γ, make combinations between P and P' through the operators *and*, *loop* and *seq* while respecting the pruning rules.

2. for each process obtained from the combinations R:
   (a) evaluate R on the subset of the log corresponding to $proj(P) \cap proj(P')$. $(evaluate(R, proj(P) \cap proj(P'))$
      – if R is frequent:
         i. add R to Θ if it satisfies the language fitness threshold. Delete the trees on the paths defined by solely constraining operators in the construction tree of R from Θ.
         ii. for each potential combination leaf a in R:
            A. create a representative word RW by replacing a with "_" in $RW(R)$
            B. if the set containing frequent trees identified by RW, $RW\Gamma$ doesn't exist, create it.
            C. call $addFreq(R, RW\Gamma)$
      – else:
         i. for each potential combination leaf a in R:
            A. create a representative word RW by replacing a with "_" in $RW(R)$
            B. if the set containing infrequent trees identified by RW, $RW\gamma$ doesn't exist, create it.
            C. call $addInfreq(R, RW\gamma)$
3. add P to Γ

Finally, we define $addInfreq(P, \gamma)$:

1. for each process tree P' in γ, make combinations between P and P' through the choice operator.
2. for each process obtained from the combinations R:
   (a) set the projection of R as $proj(P) \cup proj(P')$ and its frequency as the size of the union. Caluclate its language fitness.
      – if R is frequent :
         i. add R to Θ if it satisfies the language fitness threshold.
         ii. for each potential combination leaf a in R:
            A. construct a representative word RW by replacing a with "_" in $RW(R)$
            B. if the set containing frequent trees identified by RW, $RW\Gamma$ doesn't exist, create it.
            C. call $addFreq(R, RW\Gamma)$
      – else:
         i. for each potential combination leaf a in R:
            A. construct a representative word RW by replacing a with "_" in $RW(R)$
            B. if the set containing infrequent trees identified by RW, $RW\gamma$ doesn't exist, create it.
            C. call $addInfreq(R, RW\gamma)$
3. add P to γ

The algorithm starts by creating the set of frequent process trees identified by the word "_"; meaning any frequent tree with a single activity can be added to it. We call the function $addFreq$ on this set for each activity in Δ. Thanks to the recursive calls in and between $addFreq$ and $addInfreq$, each time we add an activity, all frequent trees composed of the activities in $_\Gamma$ are found. We can use a maximum recursion depth d and therefore a maximum depth for the found trees to force termination. Finally, we find in Θ maximal relevant patterns defined in the context of the depth d.

7 Case Study

COBPAM has been implemented as a plugin in the ProM framework [9]. We used our method to extract insights for a log representing the treatment process for Sepsis cases in a hospital. It contains 1050 cases and a total number of 15214 events recorded for 16 different activities. Fig. 3 depicts the end-to-end Petri net model characterizing the log obtained with the FHM algorithm. The analysis of this model shows that the domain is unstructured with intertwined execution paths and a high number of choice constructs, loops and edges.

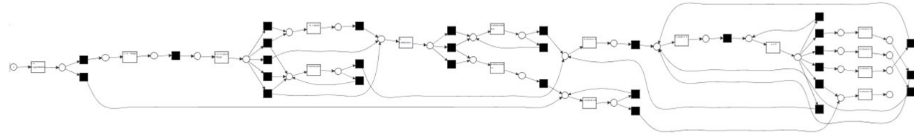

Fig. 3: The Petri net resulting from applying the FHM algorithm on the Sepsis cases log

We ran the algorithm for LPM discovery [26] on a machine with an i5-1.8 Ghz processor. We leave the default parameters. Note that the number of discovered trees can't be more than 500. In this case, exactly 500 were discovered. We show some of them that exceed a frequency threshold of 0.7 and a language fitness threshold of 0.7 in Fig. 4. We also executed COBPAM on the same machine. The maximal depth was 2 and 386 maximal relevant patterns were found with a frequency threshold of 0.7 and a language fitness threshold of the same value. We select a few in Fig. 5. We notice the difference in the trees derived by the two algorithms. For instance, the tree (12) was not extracted by COBPAM, because it is not maximal. In fact, it is contained in another frequent tree which is (17). Knowing that (17) is frequent gives the knowledge that the behavior: *"ER Registration"* followed by *"CRP"* followed by *"Leucocytes"* is frequent and as such the behavior *"CRP"* followed by *"Leucocytes"* depicted in (12) is frequent too. Other trees that are not discovered by COBPAM include loops over one activity since it is an unary operator.

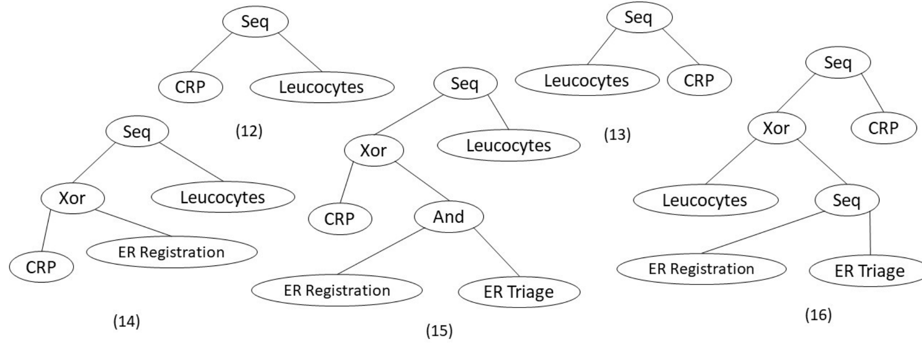

Fig. 4: Behavioral patterns mined with LPM discovery algorithm

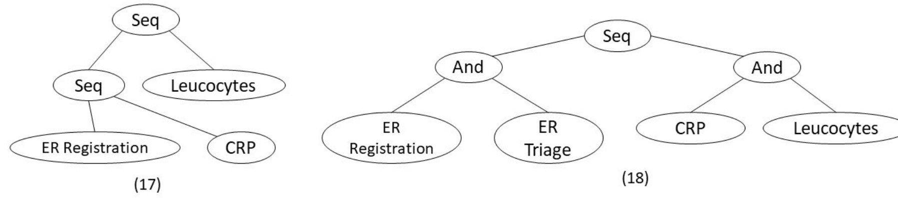

Fig. 5: Behavioral patterns mined with COBPAM using a threshold of 0.7

The trees (14), (15) and (16) are not discovered by COBPAM either since they are deemed irrelevant. Trees (14) and (15) can be obtained from (12) when replacing the activity *"CRP"* by a choice including the same activity. Since it's a choice between *"CRP"* and another behavior knowing that the tree (12) is frequent then the trees (14) and (15) are surely frequent. They don't give any new information and we don't know if the path *seq(ER Registration, Leucocytes)* is frequent in (14). The same with the tree (16) that can be constructed out of (13). The construction and evaluation (computation of support and language fitness) of such irrelevant trees is a time wasted that can be used to mine more complex process trees such as the pattern (18) presented by COBPAM. A pattern that brings new information and confirms that the path *seq(ER Registration, Leucocytes)* along with other paths is frequent. When analyzing patterns generated with COBPAM, any path traversing a *xor* operator is known to be infrequent which is a piece of information itself.

In order to compare run-times, we executed the LPM discovery algorithm and COBPAM with the same parameters on two other logs of unstructured processes [3]. A first of a real life dutch hospital containing 1143 traces, 150291 events and 624 activities and a second of an information system managing road traffic fines containing 150370 traces, 561470 events and 11 activities. The execution times are given in Table 1. They depend on the size of the log, the number of activities

and events and the complexity of the behavioral patterns existing in the log. Globally, COBPAM performs better.

Table 1: Execution time of LPM Discovery and COBPAM on different logs

	Sepsis Cases	Traffic Fines	Hospital
COBPAM	22m	28m	6h
LPM discovery	88m	68m	>48h

8 Conclusion and future works

In this paper, we proposed COBPAM, an efficient algorithm for behavioral pattern mining. Potential patterns are obtained by combining simpler patterns using algebraic operations on process trees. Compared with an exhaustive search, the efficiency of the algorithm is improved by pruning the search space, evaluating the candidates solely on parts of the event log (using projections) and exploiting calculations already done for smaller trees. Moreover, the algorithm exploits a partial order on potential patterns to discover only those that are maximal. A case study with real-world data containing logs of unstructured processes demonstrated how COBPAM improves over the state-of-the-art in behavioral pattern mining in terms of execution time and relevance of extracted patterns.

In future works, we plan to investigate the relationship between operators. For instance, if *and(a,b)* is infrequent then *seq(a,b)* is surely infrequent. That is because the behavior of the sequence operator is included in that of the concurrence operator. On another hand, while the current implementation of the algorithm is able to extract small patterns (3-4 activities) in reasonable time, in order to be able to handle huge volume of logs and output maximal patterns of any size, we will investigate using parallel computing frameworks.

References

1. Van der Aalst, W.: Process mining: Data science in action. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
2. Adriansyah, A.: Aligning Observed and Modeled Behavior. Ph.D. thesis (2014)
3. Augusto, A., Conforti, R., Dumas, M., La Rosa, M., Maggi, F.M., Marrella, A., Mecella, M., Soo, A.: Automated Discovery of Process Models from Event Logs: Review and Benchmark (2018)
4. Bose, R.P.J.C., van der Aalst, W.M.: Context Aware Trace Clustering: Towards Improving Process Mining Results. In: Proceedings of the 2009 SIAM International Conference on Data Mining (2009)
5. Bose, R.P.C., Van Der Aalst, W.M.: Trace clustering based on conserved patterns: Towards achieving better process models. In: Lecture Notes in Business Information Processing (2010)

6. vanden Broucke, S.K., De Weerdt, J.: Fodina: A robust and flexible heuristic process discovery technique. Decision Support Systems **100**, 109–118 (8 2017)
7. Buijs, J.C., Van Dongen, B.F., Van Der Aalst, W.M.: A genetic algorithm for discovering process trees. In: 2012 IEEE Congress on Evolutionary Computation, CEC 2012. pp. 1–8. IEEE (6 2012)
8. Conforti, R., Dumas, M., García-Bañuelos, L., La Rosa, M.: BPMN Miner: Automated discovery of BPMN process models with hierarchical structure. Information Systems **56**, 284–303 (3 2016)
9. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The ProM Framework: A New Era in Process Mining Tool Support. pp. 444–454. Springer, Berlin, Heidelberg (2005)
10. Fournier-Viger, P., Chun, J., Lin, W., Kiran, R.U., Koh, Y.S., Thomas, R.: A Survey of Sequential Pattern Mining. Ubiquitous International **1**(1), 54–77 (2017)
11. Greco, G., Guzzo, A., Pontieri, L., Saccà, D.: Discovering expressive process models by clustering log traces. IEEE Transactions on Knowledge and Data Engineering (2006)
12. Günther, C.W., van der Aalst, W.M.P.: Fuzzy Mining – Adaptive Process Simplification Based on Multi-perspective Metrics. pp. 328–343. Springer, Berlin, Heidelberg (2007)
13. ISO: ISO/IEC 19505-1: 2012 Information technology — Object Management Group Unified Modeling Language (OMG UML) — Part 1: Infrastructure (2012), https://www.iso.org/standard/52854.html
14. Leemans, M., van der Aalst, W.M.: Discovery of frequent episodes in event logs. In: Lecture Notes in Business Information Processing. vol. 237, pp. 1–31. Springer, Cham (11 2015)
15. Leemans, S.J.J., Fahland, D., Van Der Aalst, W.M.P.: LNCS 7927 - Discovering Block-Structured Process Models from Event Logs - A Constructive Approach pp. 311–329 (2013)
16. Maggi, F.M., Mooij, A.J., Van Der Aalst, W.M.: User-guided discovery of declarative process models. In: IEEE SSCI 2011: Symposium Series on Computational Intelligence - CIDM 2011: 2011 IEEE Symposium on Computational Intelligence and Data Mining. pp. 192–199. IEEE (4 2011)
17. Object Management Group: Notation (BPMN) version 2.0. OMG Specification https://www.omg.org/spec/BPMN/2.0/About-BPMN/
18. Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.C.: Mining sequential patterns by pattern-growth: The prefixspan approach. IEEE Transactions on Knowledge and Data Engineering (2004)
19. Reisig, W., Wolfgang: Petri nets : an introduction. Springer Berlin Heidelberg (1985)
20. Scheer, A.W., Thomas, O., Adam, O.: Process Modeling using Event-Driven Process Chains. In: Process-Aware Information Systems: Bridging People and Software through Process Technology (2005)
21. Senderovich, A., Weidlich, M., Gal, A.: Temporal Network Representation of Event Logs for Improved Performance Modelling in Business Processes. BPM **1**, 3–21 (2017)
22. Solé, M., Carmona, J.: Process mining from a basis of state regions. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 6128 LNCS, pp. 226–245. Springer, Berlin, Heidelberg (2010)
23. Song, M., Günther, C.W., Van Der Aalst, W.M.: Trace clustering in process mining. In: Lecture Notes in Business Information Processing (2009)

24. Srikant, R., Agrawal, R.: Mining sequential patterns: Generalizations and performance improvements. pp. 1–17. Springer, Berlin, Heidelberg (1996)
25. Tax, N., Dalmas, B., Sidorova, N., van der Aalst, W.M., Norre, S.: Interest-driven discovery of local process models. Information Systems **77**, 105–117 (9 2018)
26. Tax, N., Sidorova, N., Haakma, R., van der Aalst, W.M.: Mining local process models. Journal of Innovation in Digital Ecosystems **3**(2), 183–196 (2016)
27. Weijters, A.J.M.M., Ribeiro, J.T.S.: Flexible heuristics miner (FHM). In: IEEE SSCI 2011: Symposium Series on Computational Intelligence - CIDM 2011: 2011 IEEE Symposium on Computational Intelligence and Data Mining. pp. 310–317 (2011)
28. van Zelst, S.J., van Dongen, B.F., van der Aalst, W.M.P.: Avoiding Over-Fitting in ILP-Based Process Discovery. pp. 163–171. Springer, Cham (2015)