2017 IEEE International Conference on Data Mining

Split Miner: Discovering Accurate and Simple Business Process Models from Event Logs

Adriano Augusto[*†], Raffaele Conforti[*], Marlon Dumas[†] and Marcello La Rosa[*]
[*]Queensland University of Technology, Australia
email: {raffaele.conforti, m.larosa}@qut.edu.au
[†]University of Tartu, Estonia
email: {adriano.augusto, marlon.dumas}@ut.ee

***Abstract*—The problem of automated discovery of process models from event logs has been intensively researched in the past two decades. Despite a rich field of proposals, state-of-the-art automated process discovery methods suffer from two recurrent deficiencies when applied to real-life logs: (i) they produce large and spaghetti-like models; and (ii) they produce models that either poorly fit the event log (low fitness) or highly generalize it (low precision). Striking a tradeoff between these quality dimensions in a robust and scalable manner has proved elusive. This paper presents an automated process discovery method that produces simple process models with low branching complexity and consistently high and balanced fitness, precision and generalization, while achieving execution times 2-6 times faster than state-of-the-art methods on a set of 12 real-life logs. Further, our approach guarantees deadlock-freedom for cyclic process models and soundness for acyclic. Our proposal combines a novel approach to filter the directly-follows graph induced by an event log, with an approach to identify combinations of split gateways that accurately capture the concurrency, conflict and causal relations between neighbors in the directly-follows graph.**

***Index Terms*—Automated Process Discovery; Process Mining; Directly-follows Graph; Event Log; BPMN;**

I. Introduction

Modern information systems maintain detailed trails of the business processes they support, including records of business process execution events, such as the creation of a case or the execution of a task within an ongoing case. Process mining techniques allow analysts to extract insights about the performance of a business process from collections of such event records, also known as *event logs* [1]. In this context, an event log consists of a set of traces, each trace itself consisting of the sequence of event records pertaining to a case.

Among other things, process mining techniques allow us to automatically discover a process model (e.g. in the standard Business Process Model and Notation – BPMN)from an event log. For it to be useful, an automatically discovered process model must accurately reflect the behavior recorded in or implied by the log. Specifically, the process model should: (i) parse the traces in the log; (ii) parse traces that are not in the log but are likely to belong to the process that produced the log; and (iii) not parse other traces. The first property is called *fitness*, the second *generalization* and the third *precision*. Moreover, the model should be as simple as possible, a property usually quantified via *complexity* measures.

Despite intensive research [1], striking a tradeoff between the above four quality dimensions (fitness, precision, generalization and complexity) has proved elusive. When applied to real-life logs, the vast majority of automated process discovery methods (e.g. the Heuristics Miner [2] and its derivatives) produce large, spaghetti-like and oftentimes behaviorally incorrect (e.g. deadlocking) process models. Another state-of-the-art method, namely the Inductive Miner [3], often produces structured and behaviorally correct process models with high fitness but poor precision – i.e. the resulting models over-generalize the behavior observed in the log.

This paper addresses this gap by proposing an automated process discovery method designed to produce simple and sound (or deadlock-free in the presence of cycles) process models, while balancing fitness, precision and generalization. The proposal combines a novel approach to filter the directly-follows graph induced by an event log, with an approach to identify combinations of split gateways that capture the concurrency, conflict and causal relations between neighbors in the directly-follows graph. Given this focus on discovering split gateways, the proposed method is named *Split Miner*.

We empirically compared Split Miner against four state-of-the-art baselines on a set of twelve real-life event logs, and using nine performance measures covering the above four quality dimensions as well as execution time.

The next section gives an overview of automated process discovery methods. Section III presents the proposed method, while Section IV discusses its evaluation. Finally, Section V draws conclusions and sketches future work directions.

II. Background and Related Work

A. Quality Dimensions in Automated Process Discovery

The quality of automatically discovered process models is generally assessed along four dimensions: *recall* (a.k.a. *fitness*), *precision*, *generalization* and *complexity* [1].

Fitness is the ability of a model to reproduce the behavior contained in the log. A fitness of 1 means that the model can reproduce every trace in the log. In this paper, we use the fitness measure proposed in [4], which measures the degree to which every trace in the log can be aligned with a trace produced by the model. *Precision* is the ability of a model to generate only the behavior found in the log. A score of 1 indicates that any trace produced by the model is contained in the log. We use the precision measure defined in [5], which is based on similar principles as the above fitness measure. Recall and precision can be combined together into a single

2374-8486/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDM.2017.9

measure of accuracy, known as F-score, which is the harmonic mean of the two measurements $\left(2 \cdot \frac{\mathit{Fitness} \cdot \mathit{Precision}}{\mathit{Fitness} + \mathit{Precision}}\right)$.

Generalization refers to the ability of a discovery method to capture behavior of the observed process that is not present in the log. To measure generalization we use k-fold cross validation. We divide the log into k parts, we discover the model from $k-1$ parts (i.e. we hold out one part), and measure the fitness of the discovered model against the holdout part, and the precision of the discovered model against the complete log.[1] This operation is repeated for every possible holdout part, and the measures are averaged, leading to a *k-fold fitness* and a *k-fold precision* measure. A k-fold fitness of 1 means that the discovered model produces traces that are part of the observed process, even if those traces are not in the log from which the model was discovered. Similarly, a k-fold precision of 1 means that the discovered model does not over-generalize the process. The F-Score computed from k-fold fitness and k-fold precision provides a single generalization measure.

Complexity quantifies how difficult it is to understand a model. Several complexity metrics have been shown to be (inversely) related to understandability [6], including *Size* (number of nodes); *Control-Flow Complexity (CFC)* (the amount of branching caused by split gateways in the model) and *Structuredness* (the percentage of nodes located directly inside a well-structured single-entry single-exit fragment).

In addition to these quality dimensions, it is natural to expect that a discovered process model is syntactically and semantically correct. A well-accepted semantic correctness property is *soundness* [7]. A BPMN model with one start and one end event is sound if: (i) any arbitrary task can be reached from the start event executing a specific sequence of tasks (no deadlocks); (ii) from any arbitrary task it is always possible reach the end event executing a specific sequence of tasks (option to complete); (iii) whenever the end event is triggered, no other tasks are still executing behind (proper completion).

B. Automated Process Discovery Methods

The α-algorithm [8] is a simple automated process discovery method based on the concept of *Directly-Follows (dependency) Graph* (DFG). In the α-algorithm, a directly-follows dependency ($a > b$) holds if an event with label a directly precedes an event with label b in at least one trace. Using this basic relation, three relations are defined: (i) *causality* ($a \rightarrow b$) if $a > b$ and $b \not> a$; (ii) *conflict* ($a\#b$) if $a \not> b$ and $b \not> a$; and (iii) *concurrency* ($a \parallel b$) if $a > b$ and $b > a$. These relations are used to discover a process model. While appealing due to its simplicity, the α-algorithm is not applicable to real-life event logs since it assumes the log to be complete (every possible trace is present) and it is too sensitive to infrequent behavior.

The *Heuristics Miner* [2] addresses these limitations and consistently performs better in terms of accuracy on incomplete and noisy logs [9]. To handle noise, the Heuristics Miner relies on a relative frequency metric between pairs of event labels, defined as $a \Rightarrow b = \left(\frac{|a>b|-|b>a|}{|a>b|+|b>a|+1}\right)$. Whenever this metric falls under a given threshold for a given pair of event labels (a,b), the directly-follows dependency $a > b$ is removed from the DFG. The filtered DFG is then used to discover splits and joins, according to heuristics defined over the frequencies of the outgoing and incoming arcs of each node. While Heuristics Miner has been shown to achieve relatively good fitness and precision in the presence of noise [9], it still outputs spaghetti-like and unsound process models when applied to large real-life event logs. *Fodina* [10] is a variant of Heuristics Miner that avoids certain types of deadlocks produced by Heuristics Miner. However, when applied to real-life event logs, Fodina produces large and often unsound models as we show later in the empirical evaluation.

Structured process models are generally more understandable than unstructured ones [11]. Moreover, structured models are sound, provided that the gateways at the entry and exit of each block match. Given these advantages, several algorithms have been designed to discover structured process models, represented for example as *process trees* [3], [12]. A process tree is a tree where each leaf is labeled with an activity and each internal node is labeled with a control-flow operator: sequence, exclusive choice, non-exclusive choice, parallelism or iteration. The *Inductive miner* [13] uses a divide-and-conquer approach to discover process trees. It first creates a DFG, filters infrequent directly-follows dependencies, and identifies cuts in the filtered DFG. A cut is a control-flow dependency along which the log can be bisected. The identification of cuts is repeated recursively, starting from the most representative one until no more cuts are found. Once all cuts are identified, the log is split into portions (one per pair of consecutive cuts) and a process tree is generated from each portion. The *Evolutionary Tree Miner (ETM)* [12] is a genetic algorithm that starts by generating a population of random process trees. At each iteration, it computes an *overall fitness* value for each tree in the population and applies mutations to a subset thereof. The algorithm iterates until a stop criterion is fulfilled, and returns the tree with highest overall fitness. Molka et al. [14] proposed another genetic discovery algorithm that produces structured models. This latter algorithm is similar in its principles to ETM, differing mainly in the set of change operations used to produce mutations. While the Inductive Miner and ETM achieve high fitness, they over-generalize the behavior observed in the log whenever the process model to be discovered is unstructured. In particular, when the Inductive Miner is unable to capture the behavioral relations in a given fragment of the DFG, it introduces a so-called "flower" structure. A flower structure involving tasks {a, b, ...} is a control-flow structure that allows tasks {a, b, ...} to be executed any number of times and in any order, hence leading to over-generalization.

The *Structured Miner* [15] addresses this limitation by relaxing the requirement of always producing a structured process model, in favor of achieving higher accuracy. Instead of directly discovering a structured model, Structured Miner first applies the Heuristics Miner to obtain an accurate but potentially unstructured or even unsound model. Next, it applies a technique to maximally structure the discovered model in combination with heuristics to simplify the model and remove unsoundness. However, the block-structuring approach of the

[1] In the empirical evaluation, we use $k = 3$ because existing measures of fitness and precision are slow to compute, making high k values impractical.

Structured Miner often fails to produce a sound process model when applied to real-life event logs as reported later.

III. Approach

Starting from a log, Split Miner produces a BPMN model in five steps (cf. Fig. 1). Like the Heuristics Miner and Fodina, the first step is to construct the DFG, but unlike these latter, Split Miner does not immediately filter the DFG. Instead, it analyzes it to detect self-loops and short-loops (which are known to cause problems in DFG-based methods) and to discover concurrency relations between pairs of tasks. In a DFG, a concurrency relation between tasks a and b shows up as two arcs: one from a to b and another from b to a, meaning that causality and concurrency are mixed up. To address this issue, whenever a likely concurrency relation between a and b is discovered, the arcs between these two tasks are pruned from the DFG. The result is called: *pruned DFG* (PDFG). In the third step, a filtering algorithm is applied on the PDFG to strike balanced fitness and precision maintaining low control-flow complexity. In the fourth step, split gateways are discovered for each task in the filtered PDFG with more than one outgoing arc. This is followed by the discovery of join gateways.

A. Directly-Follows Graph and Short-Loops Discovery

Split Miner takes as input an event log defined as follows.

Definition 1 (Event Log): Given a set of events $\mathscr{E}$, an *event log* $\mathscr{L}$ is a set of traces as $\mathscr{T}$, where a trace $t \in \mathscr{T}$is a sequence of events $t = \langle e_1, e_2, \dots, e_n \rangle$, with $e_i \in \mathscr{E}, 1 \leq i \leq n$. Additionally, each event has a label $\mathrm{l} \in L$ and it refers to a task executed within a process, we retrieve the label of an event with the function $\lambda : \mathscr{E} \to L$, using the notation $\lambda(e) = e^l$.

Given the set of labels $L = \{a,b,c,d,e,f,g,h\}$, a possible log is: $\mathscr{L} = \{\langle a,b,c,g,e,h\rangle^{10} \langle a,b,c,f,g,h\rangle^{10} \langle a,b,d,g,e,h\rangle^{10}_{10}$ $\langle a,b,d,e,g,h\rangle^{10} \langle a,b,e,c,g,h\rangle^{10} \langle a,b,e,d,g,h\rangle^{10} \langle a,c,b,e,g,h\rangle$ $\langle a,c,b,f,g,h\rangle^{10} \langle a,d,b,e,g,h\rangle^{10} \langle a,d,b,f,g,h\rangle^{10}\}$; this log contains 10 distinct traces, each of them recorded 10 times.

Starting from a log, we construct a DFG in which each arc is annotated with a frequency, based on the following definitions.

Definition 2 (Directly-Follows Frequency): Given an event log $\mathscr{L}$, and two events labels $\mathrm{l}_1, \mathrm{l}_2 \in L$, the directly-follows frequency between l_1 and l_2 ($|\mathrm{l}_1 \to \mathrm{l}_2|$) is $| \{(e_i, e_j) \in \mathscr{E} \times \mathscr{E} \mid e_i^l = \mathrm{l}_1 \wedge e_j^l = \mathrm{l}_2 \wedge \exists t \in \mathscr{L}[\exists e_x \in t[e_x = e_i \wedge \ e_{x+1} = e_j]]\} |$.

Definition 3 (Directly-Follows Graph): Given an event log $\mathscr{L}$, its *Directly-Follows Graph (DFG)* is a directed graph $\mathscr{G} = (T, E)$, where T is the non-empty set of tasks, for which exists a bijective function $l : T \mapsto L$, where t^{l} retrieve the label of t, and E is the set of edges $E = \{(a,b) \in T \times T \mid \left|a^{\mathrm{l}} \to b^{\mathrm{l}}\right| > 0\}$. Moreover, given a task $t \in T$ we use the operator $\bullet t = \{(a,b) \in E \mid b = t\}$ and $t \bullet = \{(a,b) \in E \mid a = t\}$ to retrieve (respectively) the set of incoming and outgoing edges.

Given the DFG, we then detect self-loops and short-loops (i.e. loops involving only one and two nodes resp.) since these are known to cause problems when detecting concurrency [8]. A self-loop exists if a task has an arc towards itself in the DFG: $|a \to a|$. Short-loops and their frequencies are detected in the log as follows.

Definition 4 (Short-Loop Frequency): Given an event log $\mathscr{L}$, and two events labels $\mathrm{l}_1, \mathrm{l}_2 \in L$, we define the number of times a short-loop pattern occurs $|a \leftrightarrow b| = |\{(e_i, e_j, e_k) \in \mathscr{E} \times \mathscr{E} \times \mathscr{E} \mid e_i^l = \mathrm{l}_1 \wedge e_j^l = \mathrm{l}_2 \wedge e_k^l = \mathrm{l}_1 \wedge \exists t \in \mathscr{L}[\exists e_x \in t[e_x = e_i \wedge e_{x+1} = e_j \wedge e_{x+2} = e_k]]\}|$, with abuse of notation we use $|a \to b|$ instead of $\left|a^{\mathrm{l}} \to b^{\mathrm{l}}\right|$ and $|a \leftrightarrow b|$ instead of $\left|a^{\mathrm{l}} \leftrightarrow b^{\mathrm{l}}\right|$.

Given two tasks a and b, a short-loop ($a \circlearrowright b$) exists iff the following conditions hold:

$$|a \to a| = 0 \quad \wedge \quad |b \to b| = 0 \tag{1}$$

$$|a \leftrightarrow b| + |b \leftrightarrow a| \neq 0 \tag{2}$$

Condition 1 guarantees that neither a nor b are in a self-loop, otherwise the short-loop evaluation may not be reliable. Indeed, if we consider a model containing a concurrency between a self-loop a and a normal task b, traces recorded during the execution of the process may contain the sub-trace $\langle a,b,a \rangle$ (which also characterize $a \circlearrowright b$). Discarding this latter case fulfilling Condition 1, we use Condition 2 to ensure $a \circlearrowright b$.

Self-loop and short-loops are trivially removed from the DFG and restored in the output BPMN model at the end. Fig. 2a shows the DFG built from the example event log $\mathscr{L}$. In this log, there are no self-loops nor short-loops to be removed.

B. Concurrency Discovery

Given a DFG, we postulate that two tasks a and b are concurrent ($a \| b$) iff three conditions hold:

$$|a \to b| > 0 \quad \wedge \quad |b \to a| > 0 \tag{3}$$

$$|a \leftrightarrow b| + |b \leftrightarrow a| = 0 \tag{4}$$

$$\frac{||a \to b| - |b \to a||}{|a \to b| + |b \to a|} < \varepsilon \quad (\varepsilon \in [0,1]) \tag{5}$$

Condition 3 captures the basic requirement for $a \| b$. Indeed, the existence of edges $e_1 = (a,b)$ and $e_2 = (b,a)$ entails that a and b can occur in any order. However, this is not sufficient to postulate concurrency since this relation may hold in three cases: (i) a and b form a short-loop; (ii) a and b are concurrent; or (iii) e_1 or e_2 occur highly infrequently and can thus be ignored. Case (i) is avoided by Condition 4. Indeed, being this latter the opposite of Condition 2, it guarantees $\neg a \circlearrowright b$. This leaves us with cases (ii) and (iii). We use Condition 5 to disambiguate between the two cases: if the condition is true we assume $a \| b$, otherwise we fall into case (iii). The intuition behind Condition 5 is that two tasks are concurrent the values of $|a \to b|$ and $|b \to a|$ should be as close as possible, i.e. both interleavings are observed with similar frequency. Therefore, the smaller is the value of ε the more balanced have to be the concurrency relations in order to be captured. Reciprocally, setting ε to 1 would catch all the possible concurrency relations.

Whenever we find $a \| b$, we remove e_1 and e_2 from E, since there is no causality but instead there is concurrency. On the other hand, if we find that either e_1 or e_2 represents infrequent behavior we remove the least frequent of the two edges. The output of this step is a *pruned DFG*.

Definition 5 (Pruned DFG): Given a *DFG* $\mathscr{G} = (T, E)$, a *Pruned DFG* (PDFG) is a connected graph $\mathscr{G}_p = (T, E_p)$, where E_p is the set of edges $E_p = E \setminus \{(a,b) \in E \mid a \| b \vee (\neg a \| b \wedge (b,a) \in E \wedge |a \to b| < |b \to a|)\}$.

In the example in Fig. 2a, we can identify four possible cases of concurrency: (b,c), (b,d), (d,e), (e,g). Setting $\varepsilon =$

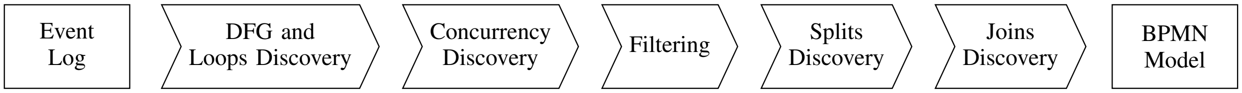
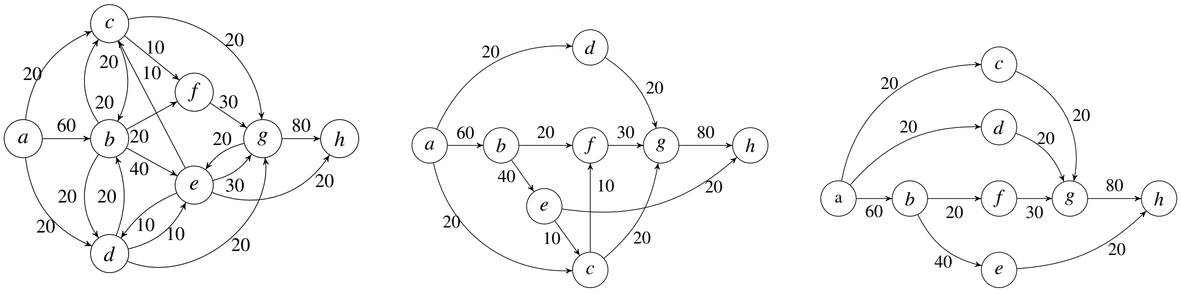


Fig. 1: Overview of the proposed approach.



(a) Initial state.

(b) After pruning.

(c) After filtering.

Fig. 2: Evolution of a directly-follows graph.

0.2, we capture the following concurrency relations: $b\|c$, $b\|d$, $d\|e$, $e\|g$. The resulting PDFG is shown in Fig. 2b.

C. Filtering

In order to derive a sound, simple, and accurate BPMN process model from a PDFG, the latter must fulfill three properties. First, each node of the PDFG must be on a path from a single start (source) to a single end (sink) node. This property is used to ensure a sound process model (no deadlocks and option to complete). Second, the number of edges of the PDFG must be minimal. This property minimizes CFC and maximizes precision, since the branching factor (used to calculate the CFC) is proportional to the number of edges, as well as the amount of allowed behavior. Third, every path from the start to the end node to have the highest possible sum of frequencies. This last property is intended to maximize fitness since the frequency of an edge is directly linked to the number of traces recording that behaviour, which then influences the number of traces we can successfully replay on the PDFG.

Unfortunately, the simultaneous fulfillment of these three properties cannot be guaranteed. For this reason, we designed a filtering algorithm for the PDFG that aims at striking a tradeoff between them. Given as input a PDFG $\mathscr{G}_p = (T, E_p)$ and a percentile η, we filter $\mathscr{G}_p$ using Algorithm 1. First, we retrieve the set F_e containing the most frequent incoming edge and the most frequent outgoing edge of each task $t \in T$ (see line 1). Second, we set a frequency threshold f_{th} as the η percentile of the frequencies of the edges in F_e, and we add to F_e all the edges with frequency greater than f_{th} (line 2 and 3). The η percentile is not taken on the frequencies of the edges in E_p because otherwise the filtering algorithm would simply retain η percentage of all the edges. Whilst our approach aims to collect in F_e only the most frequent edges of the PDFG. Next, we restore some of the most frequent edges as follows. We select the most frequent edge (e_m) in F_e (see line 6), and restore it iff its frequency is above f_{th} *or* its source post-set is empty *or* its target pre-set is empty (line 7). Finally, we remove the edge from the set of frequent edges (see line 8). This operation is repeated until all the most frequent edges are analyzed.

This filtering strikes a tradeoff between the second and third property, by retaining the minimum number of edges such that all most frequent edges (with frequency over f_{th}) are kept and each task is connected to its must frequent successor and predecessor. However, this algorithm does not guarantee that a filtered PDFG fulfills the first property. This is guaranteed *a posteriori* by removing all the tasks that after the filtering are unreachable via a forward (or backward) exploration starting from the start (or end) task. Intuitively, the removed tasks fall on infrequent traces of the event log. Also, this filtering approach relies on a simple threshold, allowing the user to balance fitness and precision. Indeed, the lower is the value of η the more edges are retained by the filter, leading to higher fitness, lower precision, and higher control-flow complexity.

Algorithm 1: Generate Filtered PDFG

```
  input: $\mathscr{G}_p = (T, E_p)$
  input: percentile $\eta$
1 Set $F_e := \{e \in E_p \mid \exists t \in T[e = getMostFrequentEdge(t\bullet) \vee e = getMostFrequentEdge(\bullet t)]\}$;
2 $f_{th} := getPercentileFrequency(F_e, \eta)$;
3 $F_e := F_e \cup \{e \in E_p \mid getFrequency(e) > f_{th}\}$;
4 $E_f = \varnothing$;
5 while $|F_e| > 0$ do
6     Edge $e_m := getMostFrequentEdge(F_e)$;
7     if $getFrequency(e_m) > f_{th} \vee |\Pi_1(e_m)\bullet| = 0 \vee |\bullet\Pi_2(e_m)| = 0$ then [2]
          $E_f := E_f \cup \{e_m\}$;
8     $F_e := F_e \setminus \{e_m\}$;
9 return $\mathscr{G}_f = (T, E_f)$;
```

Fig. 2c shows the filtered PDFG obtained as output of the previous step (Fig. 2b). In this example we removed two edges: (e,c), (c,f), both of frequency 10. Regardless the value of η we will not retain any of these latter.

D. Splits Discovery

In this step, we add split gateways into the DFG in order to capture choice and concurrency. This is the first step towards converting a DFG into a BPMN process model.

A *BPMN process model* is a connected graph $\mathscr{M} = (i, o, T, G^{+}, G^{\times}, G^{\circ}, E_m)$, where T is a non-empty set of tasks,

[2] Π is the projection operator over tuples.

Algorithm 2: Discover Splits

```
   input: Filtered PDFG $\mathscr{G}_f = (T, E_f)$, Task $a$
1  Set $K := getSuccessorTasks(a)$;
2  Function $C := \{(t, \varnothing) \in N \times 2^N \mid t \in K\}$;
3  Function $F := \{(t, \varnothing) \in N \times 2^N \mid t \in K\}$;
4  for $s_1 \in K$ do
5      Set $C_{s_1} := \{s_1\}$;
6      Set $F_{s_1} := \varnothing$;
7      for $s_2 \in K$ do
8          if $(s_2 \neq s_1 \wedge s_2 \| s_1)$ then $F_{s_1} := F_{s_1} \cup \{s_2\}$;
9      $C := C \oplus \{(s_1, C_{s_1})\}^3$;
10     $F := F \oplus \{(s_1, F_{s_1})\}$;
11 $E_m := E_f \setminus a\bullet$;
12 BPMN $\mathscr{M} := (i, o, T, \varnothing, \varnothing, \varnothing, E_m)$;
13 while $|K| > 1$ do
14     discoverXORsplits($\mathscr{M}, K, C, F$);
15     discoverANDsplit($\mathscr{M}, K, C, F$);
16 $\mathscr{M} := (i, o, T, G^+, G^\times, G^\circ, E_m \cup \{(t, k) \in N \times N \mid t = a \wedge k \in K\})$;
```

Algorithm 3: Discover XOR-splits

```
   input: BPMN $\mathscr{M}$, Set $K$, Function $C$, Function $F$
1  do
2      Set $X := \varnothing$;
3      for $k_1 \in K$ do
4          Set $C_u := C(k_1)$;
5          Set $F_s := F(k_1)$;
6          Set $F_{k_1} := F(k_1)$;
7          for $k_2 \in K$ do
8              Set $F_{k_2} := F(k_2)$;
9              if $F_{k_1} = F_{k_2} \wedge k_1 \neq k_2$ then
10                 $X := X \cup \{k_2\}$;
11                 $C_u := C_u \cup C(k_2)$;
12         if $X \neq \varnothing$ then
13             $X := X \cup \{k_1\}$;
14             break;
15     if $X \neq \varnothing$ then
16         Gateway $xor := newXOR()$;
17         $G^\times := G^\times \cup \{xor\}$;
18         $E_m := E_m \cup \{(g, x) \in N \times N \mid g = xor \wedge x \in X\}$;
19         $C := C \oplus \{(x, \varnothing) \in N \times 2^N \mid x \in X\}$;
20         $F := F \oplus \{(x, \varnothing) \in N \times 2^N \mid x \in X\}$;
21         $C := C \oplus \{(x, c) \in N \times 2^N \mid x = xor \wedge c \in C_u\}$;
22         $F := F \oplus \{(x, f) \in N \times 2^N \mid x = xor \wedge f \in F_s\}$;
23         $K := (K \cup \{xor\}) \setminus X$;
24 while $X \neq \varnothing$ ;
```

i is the start event, o is the end event, G^+ is the set of AND-gateways, $G^\times$ is the set of XOR-gateways, G° is the set of OR-gateways, and $E_m \subseteq (N \setminus \{o\}) \times (N \setminus \{i\})$ is the set of edges, where $N = \{i\} \cup \{o\} \cup T \cup G^+ \cup G^\times \cup G^\circ$ is the set of nodes. Each $g \in (G^+ \cup G^\times \cup G^\circ)$ can be a *split* or a *join* gateway. A split is a gateway with one incoming edge and multiple outgoing edges. A join is a gateway with multiple incoming edges and one outgoing edge.

To generate the split gateways, we rely on the concurrency relations identified during the second step of our approach (section III-B). With the help of the example in Figure 2c we will explain how Algorithm 2 generates a hierarchy of splits.

Starting from the filtered PDFG and a task with multiple outgoing edges, e.g. a, our algorithm retrieves all nodes following a, a.k.a. its *successors* (line 1). For each successor, e.g. b, we detect its *concurrent future* (hereafter *future*). The *future* of a successor are all the other successors for which hold a concurrency relation. In our example, we can detect d and c as future of b, since we identified $b \| c$ and $b \| d$ (in section III-B). Further, we define the concept of *cover* of a successor. If a successor is a task, its *cover* is the task itself, instead, if a successor is a gateway its cover is the set of the tasks that can be traversed only after the traversal of the gateway. In our example, at the beginning, all the successors of a are nodes, so that each of them is the *cover* of itself, e.g. b is the *cover* of b. The idea behind the algorithm is that mutually exclusive successors of a given task must share the same concurrency relations (*future*). On the other hand, successors sharing the union of their *covers* and *futures* are meant to be executed concurrently.

After computing *cover* and *future* of each successor (Algorithm 2, line 4 to 10), we use them to discover XOR-splits and AND-splits in two phases (resp. lines 14 and 15). In the first phase – Algorithm 3 – we look for all the successors sharing the same *future* (line 9). Whenever we find two or more (line 12), we introduce an XOR-split which proceed these successors, taking their place as successor of the input task (i.e. a). This gateway has as *future* the shared *future* of the selected successors, and as *cover* the union of their *covers*.

We repeat this operation until no further XOR-splits are identified (line 24). Once all possible XOR-splits are discovered, we move toward the second phase, i.e. the discovery of an AND-split – Algorithm 4. Unlike the XOR-split, we introduce an AND-split when we identify a set of successors sharing the union of their *cover* and *future* (see lines 8 and 12). The newly introduced AND-split has as *future* the intersection of the *futures* of the set of the selected successors and as *cover* the union of their *covers*. The AND-split becomes a new successor for the input task. We repeat these two steps until input task a has only one successor (Algorithm 2, line 13).

Initialization		
Key	**Cover**	**Future**
b	b	c,d
c	c	b
d	d	b
after 1^{st} iteration of Algorithm 2		
b	b	c,d
xor	c, d	b
after 2^{nd} iteration of Algorithm 2		
and	b, c, d	-

TABLE I: Splits discovery example.

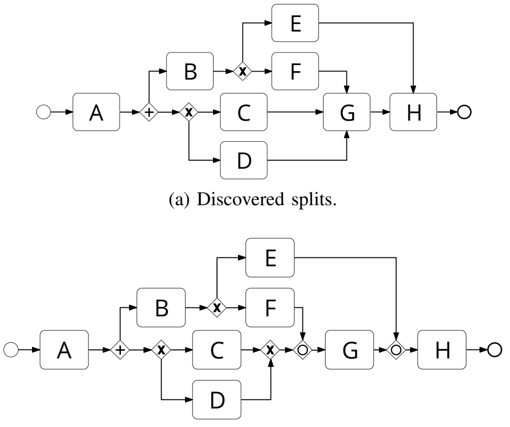
Table I shows how set K and functions C and F evolve when applying Algorithm 2 on the example in Figure 2c (given as input task a). Starting from the status depicted in the top part of the table (see initialization), after the first iteration we discover the *xor* preceding tasks c and d, since these two tasks share the same concurrency relation with b, i.e. $c \| b$, $d \| b$. Successively, after the second iteration we detect the *and* preceding task b and the *xor*, which leaves task a with just one successor, fulfilling the stop criterion of the algorithm. Figure 3a shows the output of this step for our example, after we ran Algorithm 2 also on task b.

E. Joins Discovery

Once all the split gateways have been placed, we can discover the join gateways. We introduce a join every time a task t has more than one incoming edge. This gateway will be the target of all incoming edges of t and it will precede t, whilst its type (XOR, AND, OR) is set according to the

[3] $\oplus$ is the override operator form the Z notation [16]

```
Algorithm 4: Discover AND-split
input: BPMN $\mathscr{M}$, Set $K$, Function $C$, Function $F$
for $k_1 \in K$ do
    Set $A := \varnothing$;
    Set $C_u := C(k_1)$;
    Set $F_i := F(k_1)$;
    Set $CF_{k_1} := C(k_1) \cup F(k_1)$;
    for $k_2 \in K$ do
        Set $CF_{k_2} := C(k_2) \cup F(k_2)$;
        if $CF_{k_1} = CF_{k_2} \wedge k_1 \neq k_2$ then
            $A := A \cup \{k_2\}$;
            $C_u := C_u \cup C(k_2)$;
            $F_i := F_i \cap F(k_2)$;
    if $A \neq \varnothing$ then
        $A := A \cup \{k_1\}$;
        break;
if $A \neq \varnothing$ then
    Gateway $and := newAND()$;
    $E_m := E_m \cup \{(g,a) \in N \times N \mid g = and \wedge a \in A\}$;
    $C := C \oplus \{(a,\varnothing) \in N \times 2^N \mid a \in A\}$;
    $F := F \oplus \{(a,\varnothing) \in N \times 2^N \mid a \in A\}$;
    $C := C \oplus \{(a,c) \in N \times 2^N \mid a = and \wedge c \in C_u\}$;
    $F := F \oplus \{(a,f) \in N \times 2^N \mid a = and \wedge f \in F_i\}$;
    $K := (K \cup \{and\}) \setminus A$;
```



(a) Discovered splits.

(b) Discovered joins.

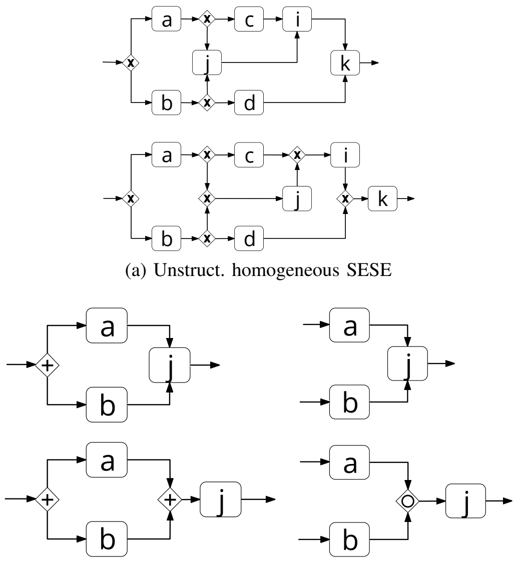
Fig. 3: Example of splits and joins discovery.

following two rules. If t is within an acyclic homogeneous Single-Entry-Single-Exit region (SESE)[4], we match the type of the entry gateway of the SESE region, otherwise the type is set to OR. These rules guarantee soundness for acyclic models and deadlock-freedom for cyclic models as discussed below.

Figure 4 shows how our approach works in case of an unstructured homogeneous SESE (Fig. 4a), in case of a structured homogeneous SESE (Fig. 4b), and in all remaining cases, i.e. (unstructured) heterogeneous SESEs (Fig. 4c).

Coming back to our working example in Fig. 3a, we detect three joins. The first one is the XOR-join at the exit of the structured SESE region including tasks c, d and g, for which the entry of the region is an XOR-split. So we match that split. The remaining two joins are within a larger heterogeneous SESE region and hence we use two OR-joins. The resulting model is shown in Fig. 3b.

Existing approaches for automated process discovery (including all those discussed in Section II) produce BPMN models with XOR and AND gateways only (no OR gateways). Also, existing quality measures for assessing the accuracy of automatically discovered process models are not designed to handle OR-joins. Accordingly, for the purposes of comparing the Split Miner with existing methods, we need to replace OR-join gateways with combinations of AND and XOR gateways. This can be achieved by the transformation proposed by Favre and Völzer [17].



(a) Unstruct. homogeneous SESE

(b) Struct. homogeneous SESE

(c) Heterogeneous SESE

Fig. 4: Examples of joins discovery.

F. Soundness and Deadlock-freedom

We sketch the proof of soundness and deadlock-freedom of the BPMN process models discovered by Split Miner. As described in section II a BPMN process model is sound if three properties are guaranteed: (i) no deadlocks, (ii) option to complete, (iii) proper completion. Property (i) is guaranteed by the rules adopted to set the types of the join gateways. Specifically, deadlocks generate from the lack of synchronization on an AND-join gateway, i.e. one or more of its incoming branches will never be active. By construction this latter case never happens because we placed AND-joins only within acyclic AND-homogenous SESE, meaning the presence of an AND-join implies the presence of one (or more) AND-split which eventually activates all the incoming branches of the AND-join. XOR-joins and OR-joins cannot generate deadlocks as discussed in [18] and [19] respectively. Property (ii) is guaranteed by our filtering algorithm. Indeed, having each task on a path from start to end, and having no deadlocks, it is trivial find the sequence of tasks that will lead to the end from any arbitrary task. Finally, property (iii) can be guaranteed only in case of acyclic process models. For acyclic process models, active branches can be left behind only if an XOR-join gateway has more than one of its incoming branches active. In such case, it fires once for each of its active incoming branches, duplicating the instance of the process execution (such that, once one instance ends, the other is still executing: no proper completion). AND-joins and OR-joins cannot leave

[4] A SESE region is said to be *homogeneous* if all its gateways are of the same type, e.g. only AND gateways. Otherwise it is heterogeneous.

branches active behind because the former waits always for all its incoming branches to be active, the latter waits for only (and all) its incoming branches which eventually will be active [18], [19]. For cyclic process models, cycles may activate again branches which already had triggered joins, generating multi-instances of the process execution and thus leading to improper completion.

G. Complexity

Let n be the number of events in the log and m be the number of tasks (distinct event labels). The DFG construction is in $O(n)$, since we sequentially read each event and generate the respective node in the graph and concurrently increase the directly-follows and short-loop frequencies. The self-loops discovery is linear on the number of nodes in the DFG, hence in $O(m)$. The short-loops discovery is done on pair of tasks, so this step is also performed in $O(m^2)$. The filtering is in $O(m^3)$, since for each task we need to find the maximum frequency over the outgoing and incoming edges. The split discovery is in $O(m^4)$, since we may run Algorithm 2 for each task, which executes m times Algorithm 3 and 4 (having two nested loops on m). The join discovery is in $O(m^4)$, since we may place a join for each two edges and defining its type is linear. Hence, the complexity of the Split Miner is in $O(n+m^4)$.

IV. Evaluation

We implemented Split Miner (hereafter SM) as a standalone Java application.[5] The tool takes as input an event log in MXML or XES format and the values for the thresholds ε and η, and it outputs a BPMN process model. Using this implementation, we empirically compared SM against five existing methods using a set of publicly available logs.

A. Datasets

We used the collection of real-life event logs available in the 4TU Centre for Research Data.[6] These logs include all *BPI Challenge* (BPIC) logs, plus other logs such as the *Road Traffic Fines Management Process* (RTFMP) and the *SEPSIS Cases* log. They record executions of business processes in a range of domains including healthcare, finance and government. We included all real-life logs of 4TU Centre except those that do not explicitly capture business processes (BPIC 2011 and 2016 logs) and the *Environmental permit application process* log, which is subsumed by BPIC 2015. In seven logs (BPIC14, the BPIC15 subset and BPIC17), we applied the filtering method in [20] to remove infrequent behavior prior to applying each of the discovery methods. Without this filtering step, all the method generated models with an F-score of close to zero due to the complexity of these logs. Table II reports the statistics of the event logs (after the initial filtering where applicable).

B. Experimental setup

We chose five state-of-the-art discover methods as baselines: Inductive Miner Infrequent (IM), Evolutionary Tree Miner (ETM), Heuristics Miner as implemented in the ProM 6 toolset (HM$_6$), Structured Miner over Heuristics Miner as implemented in ProM 6 (S-HM$_6$), and Fodina Miner (FO).

Log Name	Total Traces	Distinct Traces	Total Events	Distinct Events	Trace Length		
					min	avg	max
BPIC12	13087	4366	262200	36	3	20	175
BPIC13$_{\text{cp}}$	1487	183	6660	7	1	4	35
BPIC13$_{\text{inc}}$	7554	1511	65533	13	1	9	123
BPIC14$_{\text{f}}$	41353	14948	369485	9	3	9	167
BPIC15$_{\text{1f}}$	902	295	21656	70	5	24	50
BPIC15$_{\text{2f}}$	681	420	24678	82	4	36	63
BPIC15$_{\text{3f}}$	1369	826	43786	62	4	32	54
BPIC15$_{\text{4f}}$	860	451	29403	65	5	34	54
BPIC15$_{\text{5f}}$	975	446	30030	74	4	31	61
BPIC17$_{\text{f}}$	21861	8767	714198	41	11	33	113
RTFMP	150370	231	561470	11	2	4	20
SEPSIS	1050	846	15214	16	3	14	185

TABLE II: Statistics of the event logs employed.

Since SM takes as input two thresholds: ε and η, we ran an exhaustive hyperparameter-optimization to identify the values leading to the highest F-score across all logs, which turned out to be $\varepsilon = 0.1$, $\eta = 0.4$. We then performed two evaluations. First, we compared all the discovery methods using their default parameters against SM with the above hyperparameter-optimized settings. Second, we hyperparameter-optimized all the baselines methods as follows.[7] IM takes as input only one threshold for noise filtering. SM, HM$_6$ (as well S-HM$_6$) and FO take as input two thresholds for filtering and balancing fitness and precision. We used steps of 0.05 (range of 0.0 to 1.0) for IM, and steps of 0.10 for the thresholds of SM, HM$_6$, S-HM$_6$ and FO. For the default parameters evaluation, we measured the quality of the produced models using all the quality dimensions discussed in Section II-A, namely fitness, precision and F-Score as proxies for accuracy, 3-fold F-score as proxy for generalization, size, CFC and structuredness (struct.) as proxies for complexity, and soundness (as defined in section II). Further, we report about the execution times. For the hyperparameter-optimized evaluation, we measured only fitness and precision, because interested to find the best balance between the two metrics. Each metric was computed on BPMN models, except for fitness, precision and soundness, which were measured on Petri nets since the measuring tools work only on Petri nets. The conversions between BPMN and Petri nets were done using ProM's *BPMN Miner* package [21].

All the tests were performed on a 6-core Xeon E5-1650 3.5Ghz with 128GB of RAM running JVM 8 with 48GB of heap space. We timed out each discovery operation from a log at 1 hour for the default parameters evaluation, and at 24 hours for the hyperparameter-optimization.[8]

C. Results

Table III shows the experimental results when using the default parameters for each method. The best score for each measure on a given log is highlighted in bold. A"-" indicates that a given accuracy or complexity measure could not be reliably computed due to syntactic or semantic issues in the discovered model (e.g. disconnected or unsound model).

[5] Available at http://apromore.org/platform/tools

[6] https://data.4tu.nl/repository/collection:event_logs_real

[7] We excluded ETM from this hyperparameter-optimization exercises due to the prohitively high execution times of this method.

[8] Results and tools for reproducibility of the experiments available at https://doi.org/10.6084/m9.figshare.5379190.v1

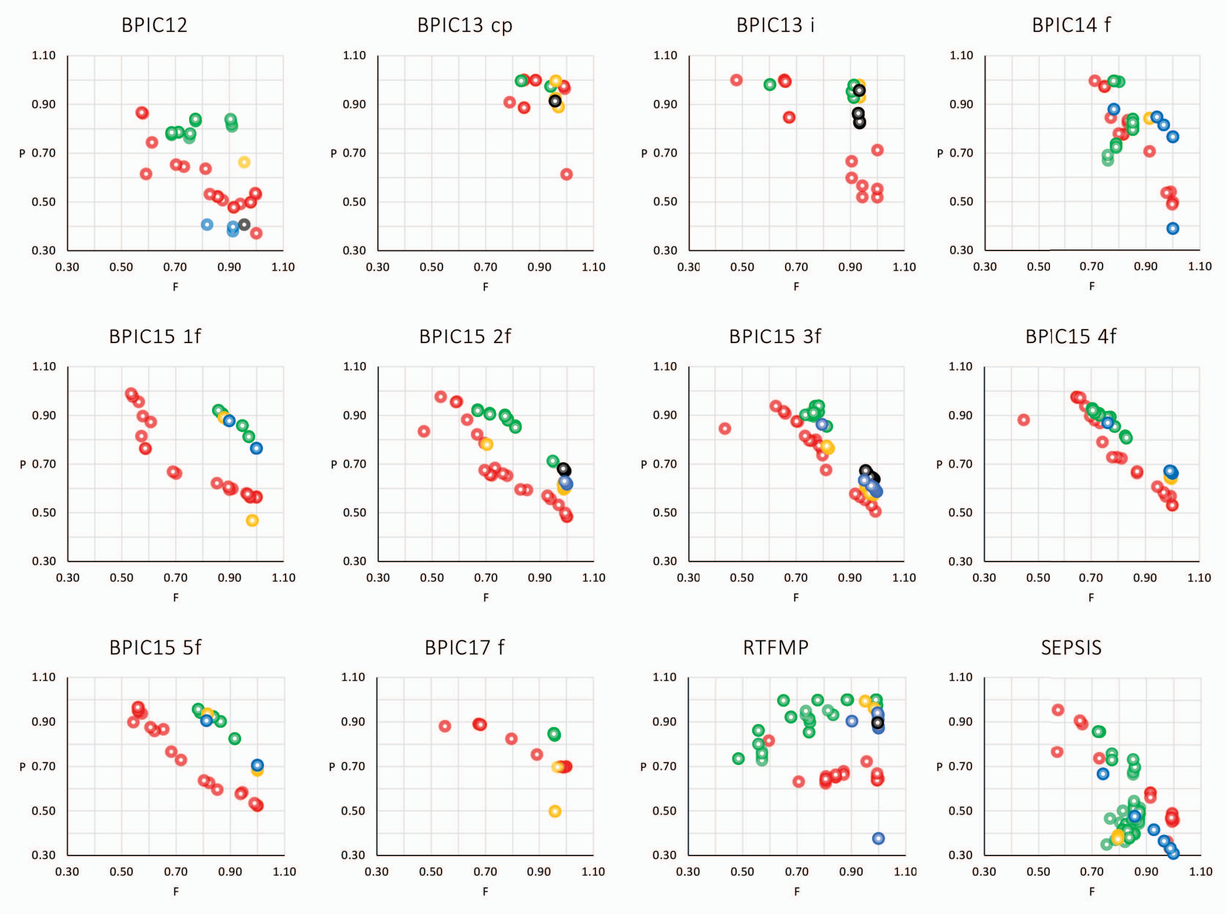


Fig. 5: Results of the hyperparameter-optimization for IM(red), SM(green), HM$_6$(black), S-HM$_6$(yellow), FO(blue). Horizontal and vertical axes report fitness and precision (respectively).

Figure 5 displays the experimental results of the experiments with hyperparameter-optimization. Each scatter plot corresponds to a log, and each dot in the scatter plot captures the fitness and precision of the model produced by a given configuration (i.e. combination of hyper-paramters) of a given method. The lack of dots corresponding to a given method on some plots (e.g. FO in BPIC13$_i$ and HM in SEPSIS) means that it was not possible to evaluate fitness or precision for the models produced by this method on the log in question.

The results in Table III put into evidence the consistently high accuracy, generalization and scalability of SM, and the low complexity of the produced models. SM strikes the best F-Score and generalization on all logs except for BPIC14$_f$ where it scores the second-highest value. But while SM excels in F-score, it generally does not achieve neither the highest fitness nor the highest precision separately. Instead, IM achieves the highest (or second-highest) fitness scores on all logs, while ETM achieves the highest precision in about half of logs. The plots of Fig. 5 show that the performance of the configurations of SM (green dots) Pareto-dominates those of other techniques in the middle ranges of fitness and precision for all logs except BPIC13$_{cp}$, BPIC13$_i$, and BPIC14$_f$. Meanwhile, IM Pareto-dominates other methods in the region with low precision and low fitness, meaning that for some configurations, IM achieves high precision at the expense of low fitness or vice-versa.

The complexity of the models discovered by SM is low, both in terms of size and CFC. And even if SM does not aim to discover structured models as opposed to IM and ETM, structuredness is generally high (over 50% in 8 logs). In those logs where SM's output is not the least complex, it is second, mostly behind ETM. Although ETM outperforms SM in complexity, the former requires very long execution times (1 hour). In contrast, SM discovers a process model in less than one second for 10 logs, being always 2-6 times faster than the second fastest method on every log.

As an example, Figs. 6 and 7 show the BPMN models discovered by IM and SM from the SEPSIS log – a log extracted from the enterprise resource planning system of a hospital, recording patient pathways in a hospital unit. We observe that the model produced by IM exhibits the "flower" pattern – all but the first activity can be skipped or repeated any number of times. This is why it achieves a fitness close to 1, but at the expense of very low precision. The model produced by SM is smaller (almost half the size), with less skipping edges and with clearly delimited loops, and is more accurate than the one produced by IM.

Log	Discovery	Accuracy			Gen.	Complexity				Exec.
Name	Method	Fitness	Precision	F-score	(3-Fold)	Size	CFC	Struct.	Sound?	Time (sec)
BPIC12	SM	0.75	0.76	**0.76**	**0.76**	**53**	32	0.72	yes	**0.58**
	IM	**0.98**	0.50	0.66	0.66	59	37	**1.00**	yes	8.34
	ETM	0.33	**0.98**	0.49	-	69	**10**	**1.00**	yes	3,600
	HM$_6$	-	-	-	-	89	197	0.28	no	2.34
	S-HM$_6$	-	-	-	-	86	46	0.20	no	370.52
	FO	-	-	-	-	102	117	0.13	no	16.03
BPIC13$_{\text{cp}}$	SM	0.94	0.97	**0.96**	**0.96**	12	7	**1.00**	yes	**0.03**
	IM	0.82	**1.00**	0.90	0.90	**9**	**4**	**1.00**	yes	0.09
	ETM	**0.99**	0.76	0.86	-	11	17	**1.00**	yes	3,600.00
	HM$_6$	-	-	-	-	13	8	-	no	0.11
	S-HM$_6$	0.94	0.93	0.94	0.95	13	**4**	**1.00**	yes	0.14
	FO	-	-	-	-	25	23	0.60	no	0.09
BPIC13$_{\text{inc}}$	SM	0.91	**0.98**	**0.94**	**0.94**	**13**	9	1.00	yes	0.23
	IM	**0.92**	0.50	0.65	0.68	**13**	**7**	**1.00**	yes	0.95
	ETM	0.84	0.80	0.82	-	28	24	**1.00**	yes	3,600.00
	HM$_6$	0.91	0.96	0.93	0.93	**13**	9	**1.00**	yes	0.84
	S-HM$_6$	0.91	**0.98**	**0.94**	**0.94**	**13**	9	**1.00**	yes	1.06
	FO	-	-	-	-	44	55	0.75	no	1.95
BPIC14$_{\text{f}}$	SM	0.76	0.67	0.71	0.75	27	16	0.74	yes	**0.59**
	IM	**0.89**	0.71	**0.79**	**0.79**	31	18	**1.00**	yes	3.45
	ETM	0.68	**0.94**	**0.79**	-	**22**	**15**	**1.00**	yes	3,600.00
	HM$_6$	-	-	-	-	44	56	-	no	4.67
	S-HM$_6$	-	-	-	-	120	51	0.27	no	16.64
	FO	-	-	-	-	37	46	0.41	no	4.79
BPIC15$_{\text{1f}}$	SM	0.90	0.88	**0.89**	**0.89**	110	43	0.50	yes	**0.48**
	IM	0.97	0.57	0.71	0.72	164	108	**1.00**	yes	1.08
	ETM	0.57	**0.89**	0.69	-	**73**	**21**	**1.00**	yes	3,600.00
	HM$_6$	-	-	-	-	150	98	-	no	1.03
	S-HM$_6$	-	-	-	-	228	122	0.57	no	139.21
	FO	**1.00**	0.76	0.87	0.86	146	91	0.26	yes	1.92
BPIC15$_{\text{2f}}$	SM	0.77	**0.90**	**0.83**	**0.82**	122	41	0.32	yes	**0.25**
	IM	0.93	0.56	0.70	0.70	193	123	**1.00**	yes	0.70
	ETM	0.62	**0.90**	0.73	-	**78**	**19**	**1.00**	yes	3,600.00
	HM$_6$	-	-	-	-	194	158	0.11	no	0.58
	S-HM$_6$	**0.98**	0.60	0.75	0.76	265	163	0.34	yes	177.27
	FO	-	-	-	-	195	159	0.09	no	2.50
BPIC15$_{\text{3f}}$	SM	0.78	0.94	**0.85**	**0.85**	90	29	0.61	yes	**0.36**
	IM	**0.95**	0.55	0.70	0.69	159	108	**1.00**	yes	1.36
	ETM	0.66	0.88	0.75	-	**78**	**26**	**1.00**	yes	3,600.00
	HM$_6$	**0.95**	0.67	0.79	0.79	157	151	0.07	yes	1.24
	S-HM$_6$	**0.95**	0.61	0.74	0.75	215	183	0.30	yes	147.97
	FO	-	-	-	-	174	164	0.06	no	2.03
BPIC15$_{\text{4f}}$	SM	0.73	0.91	**0.81**	**0.80**	96	31	0.31	yes	**0.25**
	IM	0.96	0.58	0.73	0.72	162	111	**1.00**	yes	1.05
	ETM	0.66	**0.95**	0.78	-	**74**	**17**	**1.00**	yes	3,600.00
	HM$_6$	-	-	-	-	158	129	0.15	no	0.55
	S-HM$_6$	**0.99**	0.65	0.78	0.78	207	137	0.29	yes	142.11
	FO	-	-	-	-	157	127	0.15	no	1.33
BPIC15$_{\text{5f}}$	SM	0.79	**0.94**	**0.86**	**0.85**	102	30	0.33	yes	**0.27**
	IM	0.94	0.18	0.30	0.61	134	95	**1.00**	yes	0.70
	ETM	0.58	0.89	0.70	-	**82**	**26**	**1.00**	yes	3,600.00
	HM$_6$	-	-	-	-	166	124	0.15	no	0.58
	S-HM$_6$	**1.00**	0.68	0.81	0.81	239	151	0.43	yes	142.28
	FO	**1.00**	0.71	0.83	0.83	166	125	0.15	yes	1.63
BPIC17$_{\text{f}}$	SM	0.95	0.85	**0.90**	**0.90**	32	17	0.75	yes	**2.53**
	IM	**0.98**	0.70	0.82	0.82	35	20	**1.00**	yes	13.91
	ETM	0.72	**1.00**	0.84	-	**31**	**5**	**1.00**	yes	3,600.00
	HM$_6$	-	-	-	-	36	18	-	no	11.81
	S-HM$_6$	0.95	0.52	0.67	0.74	42	13	**1.00**	yes	10.31
	FO	-	-	-	-	98	82	0.25	no	70.58
RTFMP	SM	0.99	**1.00**	**1.00**	**1.00**	**22**	**16**	0.46	yes	**1.25**
	IM	0.99	0.63	0.77	0.80	34	20	**1.00**	yes	7.92
	ETM	0.79	0.98	0.87	-	46	33	**1.00**	yes	3,600.00
	HM$_6$	-	-	-	-	47	51	0.13	no	9.42
	S-HM$_6$	0.98	0.95	0.96	0.96	163	97	**1.00**	yes	274.44
	FO	**1.00**	0.94	0.97	0.97	31	32	0.19	yes	6.25
SEPSIS	SM	0.73	**0.86**	**0.79**	**0.80**	31	20	0.97	yes	**0.05**
	IM	**0.99**	0.48	0.65	0.61	50	32	**1.00**	yes	0.30
	ETM	0.71	0.84	0.77	-	**30**	**15**	**1.00**	yes	3,600.00
	HM$_6$	-	-	-	-	82	137	0.17	no	0.16
	S-HM$_6$	0.92	0.42	0.58	0.58	225	131	**1.00**	yes	322.88
	FO	-	-	-	-	60	63	0.28	no	0.27

TABLE III: Evaluation results.

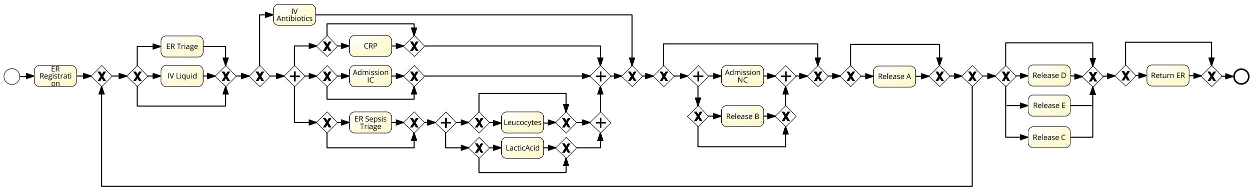


Fig. 6: Model discovered by IM from the Sepsis log.

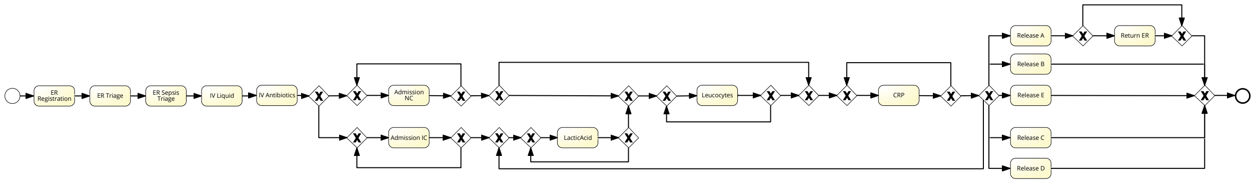


Fig. 7: Model discovered by SM from the Sepsis log.

V. Conclusion and Future Work

Split Miner is a step forward towards more scalable and robust methods for automated process discovery. Split Miner outperformed all baselines in terms of F-score and generalization in 11 out of 12 real-life logs, it produced smaller models than all baselines except Evolutionary Tree Miner, and its execution times were 2-6 times faster than the closest baseline.

One of the keystones of Split Miner is a filtering method for directly-follows graphs. The proposed method however only filters directly-follows relations (not tasks) and thus an additional preprocessing filter was required to handle the BPIC14, BPIC15 and BPIC17 logs (the same filter was required by all baselines). A possible avenue for future work is to design a filtering approach combining the strengths of the preprocessing filter used in the experiments with Split Miner's filter.

A second keystone of Split Miner is its ability to discover combinations of split gateways that capture the behavioral relations between a given task and its successors in the directly-follows graph. On the other hand, discovering the corresponding join gateways is a challenge that is only partially addressed in this proposal. The approach employed to identify situations where an OR-join can be directly replaced by an AND-join or an XOR-join can be refined, as we do not guarantee a minimal use of the OR-join. Finally, Split Miner does not guarantee sound models in all cases, though it ensures always deadlock-freedom. Addressing these limitations is a direction for future work.

Reproducibility. Links to all tools and datasets required to reproduce the experiments are given in Sections IV.B-IV.C.

Acknowledgments. This research was funded by the Australian Research Council (grant DP150103356) and the Estonian Research Council (grant IUT20-55)

References

[1] W. van der Aalst, *Process Mining - Data Science in Action*. Springer, 2016.

[2] A. Weijters and J. Ribeiro, "Flexible Heuristics Miner (FHM)," in *Proc. of CIDM*. IEEE, 2011.

[3] S. Leemans, D. Fahland, and W. van der Aalst, "Discovering block-structured process models from event logs - a constructive approach," in *Proc. of Petri Nets*, ser. LNCS. Springer, 2013.

[4] A. Adriansyah, B. van Dongen, and W. van der Aalst, "Conformance checking using cost-based fitness analysis," in *Proc. of EDOC*. IEEE, 2011.

[5] A. Adriansyah, J. Muñoz-Gama, J. Carmona, B. van Dongen, and W. van der Aalst, "Measuring precision of modeled behavior," *ISeB*, vol. 13, no. 1, pp. 37–67, 2015.

[6] J. Mendling, *Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness*. Springer, 2008.

[7] W. van der Aalst, K. van Hee, A. ter Hofstede, N. Sidorova, H. Verbeek, M. Voorhoeve, and M. Wynn, "Soundness of workflow nets: classification, decidability, and analysis," *Formal Asp. Comput.*, vol. 23, no. 3, 2011.

[8] W. van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 9, 2004.

[9] J. D. Weerdt, M. D. Backer, J. Vanthienen, and B. Baesens, "A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs," *Inf. Syst.*, vol. 37, no. 7, 2012.

[10] S. K. vanden Broucke and J. De Weerdt, "Fodina: a robust and flexible heuristic process discovery technique," *Decision Support Systems*, 2017.

[11] M. Dumas, M. La Rosa, J. Mendling, R. Mäesalu, H. Reijers, and N. Semenenko, "Understanding business process models: the costs and benefits of structuredness," in *Proc. of CAiSE*. Springer, 2012.

[12] J. Buijs, B. van Dongen, and W. van der Aalst, "On the role of fitness, precision, generalization and simplicity in process discovery," in *Proc. of CoopIS*, ser. LNCS 7565. Springer, 2012.

[13] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, *Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour*. Cham: Springer International Publishing, 2014, pp. 66–78.

[14] T. Molka, D. Redlich, W. Gilani, X. Zeng, and M. Drobek, "Evolutionary computation based discovery of hierarchical business process models," in *Proc. of BIS*. Springer, 2015.

[15] A. Augusto, R. Conforti, M. Dumas, M. L. Rosa, and G. Bruno, "Automated discovery of structured process models: Discover structured vs. discover and structure," in *Proc. of ER*, ser. LNCS 9974. Springer, 2016.

[16] A. Diller, *Z: An introduction to formal methods*. John Wiley & Sons, Inc., 1990.

[17] C. Favre and H. Völzer, "The difficulty of replacing an inclusive or-join," in *Proc. of BPM*, ser. LNCS 7481. Springer, 2012.

[18] R. M. Dijkman, M. Dumas, and C. Ouyang, "Semantics and analysis of business process models in {BPMN}," *Information and Software Technology*, vol. 50, no. 12, pp. 1281 – 1294, 2008.

[19] H. Völzer, "A new semantics for the inclusive converging gateway in safe processes," in *International Conference on Business Process Management*. Springer, 2010, pp. 294–309.

[20] R. Conforti, M. L. Rosa, and A. ter Hofstede, "Filtering out infrequent behavior from business process event logs," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 2, 2017.

[21] R. Conforti, M. Dumas, L. García-Bañuelos, and M. La Rosa, "BPMN Miner: Automated discovery of BPMN process models with hierarchical structure," *Inf. Syst.*, vol. 56, 2016.