Noname manuscript No.
(will be inserted by the editor)

Split Miner: Automated Discovery of Accurate and Simple Business Process Models from Event Logs

Adriano Augusto · **Raffaele Conforti** · **Marlon Dumas** · **Marcello La Rosa** · **Artem Polyvyanyy**

Received: 22 Dec 2017 / Revised: 09 Apr 2018 / Accepted: 27 Apr 2018

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 over-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, namely Split Miner, which produces simple process models with low branching complexity and consistently high and balanced fitness and precision, while achieving considerably faster execution times than state-of-the-art methods, measured on a benchmark covering twelve real-life event logs. Split Miner 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. Split Miner is also the first automated process discovery method that is guaranteed to produce deadlock-free process models with concurrency, while not being restricted to producing block-structured process models.

Keywords Process Mining, Automated Process Discovery, Event Log, BPMN

1 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* [30]. In this context, an

Adriano Augusto · Raffaele Conforti · Marcello La Rosa · Artem Polyvyanyy
The University of Melbourne
E-mail: {raffaele.conforti,marcello.larosa,artem.polyvyanyy}@unimelb.edu.au

Adriano Augusto · Marlon Dumas
University of Tartu
E-mail: {adriano.augusto,marlon.dumas}@ut.ee

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 business process model (e.g. in the standard Business Process Model and Notation – BPMN[1]) from an event log. For it to be useful, an automatically discovered process model ought to accurately reflect the behavior recorded in or implied by the event 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 one *generalization* and the third one *precision*. Moreover, the model should be as simple as possible, a property usually quantified via *complexity* measures.

Despite intensive research [35,5], 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, such as the Heuristics Miner [36] 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 [16], often produces block-structured and hence behaviorally correct process models with high fitness, but poor precision. In other words, the produced process models over-generalize the behavior observed in the event log.

This article addresses this gap by proposing an automated process discovery method designed to produce simple 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*.

In its basic variant, Split Miner generates BPMN process models with OR-join gateways. Empirical studies have shown that OR-join gateways induce higher cognitive overload on process model users than the alternative AND-join and XOR-join gateways, which have a simpler semantics. Hence, well-accepted guidelines for process modeling recommend that OR-join gateways should be used sparingly [22]. Accordingly, Split Miner incorporates an algorithm to replace OR-join gateways with AND-join or XOR-join gateways, while guaranteeing that the resulting process model remains sound (in the case of acyclic process models), and deadlock-free in all cases. To the best of our knowledge, the Split Miner is the first automated process discovery method that produces process models that are guaranteed to be deadlock-free, while not being restricted to producing block-structured process models only.

The article also reports on an empirical comparison between Split Miner and four state-of-the-art baselines based 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.

This article is an extended and revised version of a conference paper [4]. With respect to the conference version, the main extensions are:

- A revised algorithm for filtering the directly-follows graph, which ensures that the filtered graph has the basic properties of a syntactically correct BPMN process model, namely that every node is on a path from a single source node to a single sink node.

[1] `http://www.bpmn.org/`

- An algorithm to post-process the output of the Split Miner in order to replace OR-join gateways with AND-join and XOR-join gateways.
- Formal proofs of the semantic properties of process models produced by the revised Split Miner algorithm.

The rest of the article is structured as follows. Section 2 provides an overview of automated process discovery methods. Next, Section 3 presents the Split Miner method, while Section 4 formally analyzes the semantic properties (deadlock-freedom and soundness) of process models produced by Split Miner. Finally, Section 5 discusses the empirical evaluation of Split Miner, while Section 6 draws conclusions and sketches future work directions.

2 Background and Related Work

This section introduces several quality dimensions and metrics for assessing the goodness of automated process discovery methods. The section also provides an overview of existing automated process discovery methods and discusses their limitations.

2.1 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* [30].

Fitness is the ability of a model to reproduce the behavior contained in the log. A fitness of one means that the model can reproduce every trace in the log. In this article, we use the fitness measure proposed in [2], which measures the degree to which each trace in the log can be aligned with a corresponding trace produced by the process model.

Precision is the ability of a model to generate only the behavior found in the log. A score of one indicates that any trace produced by the process model is contained in the log. We use the precision measure defined in [1], which is based on similar principles as the above fitness measure. Recall and precision can be combined together into a single measure of accuracy, known as F-score, which is the harmonic mean of fitness and precision, i.e. $\left(2 \cdot \frac{Fitness \cdot Precision}{Fitness + Precision}\right)$.

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.[2] 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 process 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 one 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 refers to how difficult it is to understand a model. Several complexity metrics have been empirically shown to be (inversely) related to the understandability

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

of process models [21]. These and other empirical findings on process model understandability are distilled in the seven Process Modeling Guidelines (7PMG) compiled by Mendling et al. [22] :

G1: *Use as few model elements as possible*; this guideline relates to the *Size* of the process model, which measures the number of nodes.
G2: *Minimize the routing paths per element*; this guideline relates to the *Control-Flow Complexity (CFC)* metric [7], which measures the amount of branching induced by the split gateways in a process model.
G3: *Use one start event for each trigger and one end event for each outcome*.
G4: *Model as structured as possible*; this guideline tells us that for every split gateway in a process model, there should be a corresponding join gateway, such that the sub-graph between the split and the join gateway is a single-entry, single-exit region. This guideline relates to the *structuredness* metric, i.e. the percentage of nodes directly located inside a single-entry single-exit fragment. The more nodes in a process model are located outside such fragments, the lower is the value of the structuredness metric.
G5: *Avoid OR gateways where possible*.
G6: *Use verb-object activity labels*.
G7: *Decompose a model with more than 30 elements*.

In the empirical evaluation reported later in the article, we include the complexity metrics that directly relate to guidelines G1, G2 and G4. Guidelines G3 and G5 are guaranteed by construction by our approach, while guidelines G6 and G7 are not applicable to our context. Indeed, G6 refers to the labeling style of activities: in our case, we take the activity labels directly from the event log. G7 only applies to methods that discover hierarchical process models, while in our case we discover flat models.

In addition to the four quality dimensions, it is natural to expect that a discovered process model is *syntactically and semantically correct*. A model is syntactically correct when all the nodes are on a path from a single start node to a single end node.[3] In other words, there are no disconnected nodes or dangling arcs. A well-accepted semantic correctness notion is *soundness* [31]. This notion has been defined on Workflow nets, and can be adapted to BPMN models as follows. A BPMN model with one start and one end event is sound if and only if: (i) no task can be enabled or executed more than once simultaneously (safeness); (ii) any arbitrary task can be reached from the start event executing a specific sequence of tasks (deadlock-freedom).

2.2 Automated Process Discovery Methods

The α-algorithm [32] 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

[3] BPMN allows process models to have multiple start and multiple end events, but such process models can be re-written as process models with a single start and a single end event, hence we can restrict ourselves to process models with a single start and a single end event without loss of generality.

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* [36] addresses these limitations and consistently performs better in terms of accuracy on incomplete and noisy logs [35]. 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 [35], it still outputs spaghetti-like and unsound process models when applied to large real-life event logs. *Fodina* [33] 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 [13, 14]. 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* [16, 6]. 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* [17] 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)* [6] 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. [23] 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* [3] 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 the technique to maximally structure the discovered model proposed in [25–27] and applies heuristics to simplify the model and remove unsound-

Fig. 1: Overview of the proposed approach.

ness. However, the block-structuring approach of the Structured Miner often fails to produce a sound process model when applied to real-life event logs as reported later.

The problem of automated process discovery is partially related to that of mining patterns from collections of sequences, also known as *sequence databases*. Classical sequential pattern mining methods extract patterns corresponding to contiguous subsequences that occur in many sequences of a sequence database [37]. More recently, algorithms have been proposed [12,29] to mine *gapped* sequential patterns – allowing gaps between two successive events of the pattern – and *repetitive* sequential patterns – which capture not only subsequences that recur in multiple sequences, but also subsequences that recur frequently within the same sequence.

In the above approaches, one pattern captures only one subsequence. In other approaches, a pattern may capture multiple subsequences, including subsequences that are not observed in the sequence database but are related to observed subsequences. In other words, the extracted patterns generalize the observed behavior, which makes these techniques closer to automated process discovery. For example, *episodes* [20, 24] extend sequential patterns with parallelism by allowing a pattern to incorporate *partial order relations*.

The authors of [18] propose a method called Post Sequential Patterns Mining (PSPM) that takes as input a set of sequential patterns and extracts concurrency relations between them. In this work, a *concurrency relation* between sequential patterns means that those patterns *occur together in the same sequences*. This notion of concurrency does not consider the case where the same pattern occurs multiple times per sequence. A later extension [19] improves this method and proposes a visual notation, called a ConSP-Graph, to represent the concurrent relations between sequential patterns.

The work in [8] extends that in [18] by extracting *exclusive relations* between sequential patterns, i.e. patterns that do not occur in the same sequences, and propose a visual graph called an ESP-graph to visually represent such relations. ConSP-Graphs and ESP-graphs can be seen as simple process models (with sequential and concurrency relations). However, while the authors of [8] mention the extension of ConSP-Graphs to a richer set of patterns as an important area of future work, no work has been done in the area of PSPM to mine graphs that can contain arbitrary combinations of concurrency, choices, sequential orderings, and loops in a single graph. This ability to deal with such arbitrary combinations of patterns sets apart automated process discovery techniques from sequential pattern mining techniques.

3 Approach

Starting from a log, Split Miner produces a BPMN model in six 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 process discovery methods) and to discover concurrency relations between pairs of tasks. In a DFG, a concurrency relation between two tasks, e.g. 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. Similarly, in the fifth step, join gateways are discovered from tasks with multiple incoming arcs. Lastly, if any OR-joins were discovered, they are removed (whenever possible).

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

For the remaining, we assume all the traces of an event log have the same start event and the same end event. This is guaranteed by a simple pre-processing of the event log, to be compliant with the third of the 7PMG (Section 2), i.e. "*use one start event for each trigger and one end event for each outcome*".

Given the set of labels $L = \{a,b,c,d,e,f,g,h\}$, a possible log is: $\mathcal{L} = \{\langle a,b,c,g,e,h\rangle^{10}, \langle a,b,c,f,g,h\rangle^{10}, \langle a,b,d,g,e,h\rangle^{10}, \langle a,b,d,e,g,h\rangle^{10}, \langle a,b,e,c,g,h\rangle^{10}, \langle a,b,e,d,g,h\rangle^{10}, \langle a,c,b,e,g,h\rangle^{10}, \langle a,c,b,f,g,h\rangle^{10}, \langle a,d,b,e,g,h\rangle^{10}, \langle a,d,b,f,g,h\rangle^{10}\}$; 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.

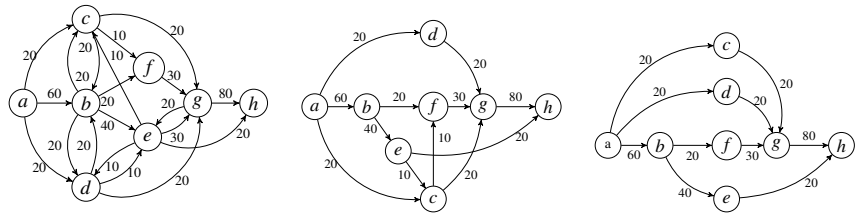
Definition 2 (Directly-Follows Frequency) Given an event log $\mathcal{L}$, and two events labels $l_1, l_2 \in L$, the directly-follows frequency between l_1 and l_2 is $|l_1 \rightarrow l_2| = |\{(e_i, e_j) \in \mathcal{E} \times \mathcal{E} \mid e_i^l = l_1 \wedge e_j^l = l_2 \wedge \exists t \in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

Definition 3 (Directly-Follows Graph) Given an event log $\mathcal{L}$, its *Directly-Follows Graph (DFG)* is a directed graph $\mathcal{G} = (N, E)$, where N is the non-empty set of nodes[4], for which exists a bijective function $l : N \mapsto L$, where n^l retrieve the label of n, and E is the set of edges $E = \{(a,b) \in N \times N \mid |a^l \rightarrow b^l| > 0\}$. Moreover, given a node $n \in N$ we use the operator $\bullet n = \{(a,b) \in E \mid b = n\}$ and $n\bullet = \{(a,b) \in E \mid a = n\}$ 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 tasks resp.) since these are known to cause problems when detecting concurrency [32]. A self-loop exists if a node has an arc towards itself in the DFG: $|a \rightarrow a|$. Short-loops and their frequencies are detected in the log as follows.

Definition 4 (Short-Loop Frequency) Given an event log $\mathcal{L}$, and two events labels $l_1, l_2 \in L$, we define the number of times a short-loop pattern occurs $|a \leftrightarrow b| = |\{(e_i, e_j, e_k) \in \mathcal{E} \times \mathcal{E} \times \mathcal{E} \mid e_i^l = l_1 \wedge e_j^l = l_2 \wedge e_k^l = l_1 \wedge \exists t \in \mathcal{L}[\exists e_x \in t[e_x = e_i \wedge e_{x+1} = e_j \wedge e_{x+2} =$

[4] Each node of the graph represents a task.



(a) Initial state. (b) After pruning. (c) After filtering.

Fig. 2: Processing of the directly-follows graph.

$e_k]\}|$, with abuse of notation we use $|a \to b|$ instead of $\left|a^1 \to b^1\right|$ and $|a \leftrightarrow b|$ instead of $\left|a^1 \leftrightarrow b^1\right|$.

Given two tasks a and b, a short-loop ($a \circlearrowleft b$) 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 $a \circlearrowleft b$). Discarding this latter case fulfilling Condition 1, we use Condition 2 to ensure $a \circlearrowleft b$.

Self-loop 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 $\mathcal{L}$. In this log, there are no self-loops nor short-loops.

3.2 Concurrency Discovery

Given a DFG and two tasks a and b, such that neither a nor b is a self-loop[5], we postulate 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 occurs highly infrequently and can thus be ignored. Case (i) is avoided by Condition 4. Since being this latter the opposite of Condition 2, it guarantees $\neg a \circlearrowleft b$. This leaves us with

[5] We favor self-loops over concurrency.

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* $\mathcal{G} = (N, E)$, a *Pruned DFG* (PDFG) is a connected graph $\mathcal{G}_p = (N, E_p)$, 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 $\varepsilon = 0.2$, we capture the following concurrency relations: $b \| c$, $b \| d$, $d \| e$, $e \| g$. The resulting PDFG is shown in Fig. 2b.

3.3 Filtering

In order to derive a sound, simple, and accurate BPMN process model from a PDFG, the latter must satisfy three properties. First, each node of the PDFG must be on a path from the single start node (source) to the single end node (sink). This property is necessary to ensure a sound process model (no deadlocks and no lack of synchronization). Second, for each node, its path from source to sink must be the one having maximum capacity. In our context, the capacity of a path is the frequency of the least frequent edge of the path. This property is meant to maximize fitness, since the capacity of a path matches the number of traces that can be replayed on that path. Third, the number of edges of the PDFG must be minimal. This property minimizes CFC and maximizes precision, since the number of edges is proportional to the branching factor (used to calculate the CFC) and to the amount of allowed behavior.

To satisfy these three properties, we designed a variant of Dijkstra's shortest path algorithm [11]. The main differences between Dijkstra's algorithm and ours are the following: (i) during the exploration of the graph we do not propagate the length of the paths, but their capacities; (ii) Dijkstra solves a problem of minimization, whilst we solve a problem of maximization. Additionally, since we want to guarantee that each node is reachable from the source and can reach the sink (i.e. on a path from source to sink), we perform a double breadth-first exploration: forward (source to sink) and backward (sink to source). During the forward exploration, for each node of the PDFG we discover its maximum source-to-node capacity (*forward capacity*), and its incoming edge granting such forward capacity (*best incoming edge*). Similarly, during the backward exploration, we discover maximum node-to-sink capacities (*backward capacities*), and outgoing edges (*best outgoing edge*). Through this algorithm we satisfy the first and second property, and we set a limit to the maximum number of edges retained in our PDFG, which is always less than $2|T|$ (i.e. each node will have at most one incoming and one outgoing edge). However, the limited number of edges may reduce the amount of behaviour that the final model can replay, and consequently its fitness. To strike a trade-off between fitness and precision, we introduce a frequency

threshold which let the user balance the two metrics. Precisely, we compute the η percentile over the frequencies of the most frequent incoming and outgoing edges of each node, and we retain those edges with a frequency exceeding the threshold. It is important to notice, that the percentile is not taken over the frequencies of all the edges in E_p since otherwise we would simply retain η percentage of all the edges.

Algorithm 1 shows in details how we achieve the objectives listed above. Given as input a PDFG $\mathcal{G}_p = (T, E_p)$ and a percentile value η, we detect the source (i) and the sink (o) of $\mathcal{G}_p$. We initialize the forward and backward capacities of each node to 0, except for the source and the sink, which are supposed infinite (line 3 to 13). Simultaneously, for each node we collect the highest frequency among its incoming edges (f_i) and the highest frequency among its outgoing edges (f_o), which are used to estimate the η percentile (line 14).

Algorithm 1: Generate Filtered PDFG

input : PDFG $\mathcal{G}_p = (T, E_p)$, percentile value η
output : Filtered PDFG $\mathcal{G}_f = (i, o, T, E_f)$

1 $i \leftarrow$ the source node of $\mathcal{G}_p$;
2 $o \leftarrow$ the sink node of $\mathcal{G}_p$;
3 Create a map $C_f : T \to \mathbb{Z}^+$;
4 Create a map $C_b : T \to \mathbb{Z}^+$;
5 Create a set $F \leftarrow \varnothing$;
6 $C_f[i] \leftarrow$ inf;
7 $C_b[o] \leftarrow$ inf;
8 **for** $t \in T$ **do**
9 $\quad C_f[t] \leftarrow 0$;
10 $\quad C_b[t] \leftarrow 0$;
11 $\quad f_i \leftarrow$ the highest frequency of the incoming edges of t;
12 $\quad f_o \leftarrow$ the highest frequency of the outgoing edges of t;
13 $\quad$ add f_i and f_o to F;
14 $f_{th} \leftarrow$ the η percentile on the values in F;
15 Create a map $E_i : T \to E_p$;
16 Create a map $E_o : T \to E_p$;
17 DiscoverBestIncomingEdges($\mathcal{G}_p$, i, C_f, E_i);
18 DiscoverBestOutgoingEdges($\mathcal{G}_p$, o, C_b, E_o);
19 Create a set $E_f \leftarrow \varnothing$;
20 **for** $e \in E_p$ **do**
21 $\quad$ **if** $(\exists t_e \mid E_i[t_e] = e \vee E_o[t_e] = e) \vee (f_e > f_{th})$ **then** add e to E_f;
22 **return** $\mathcal{G}_f = (T, E_f)$;

After the initialization, we perform the breadth-first forward exploration of $\mathcal{G}_p$ starting from the source i. We use a queue (Q) in order to store the nodes to explore, which at the beginning contains only i, and a set (U) for the unexplored nodes containing all the nodes in T (except for i). Finally, the map E_i will store the best incoming edge of each visited node.

Using a FIFO[6] policy, we start the exploration of the nodes in Q. When a node p is removed from Q, we analyse its outgoing edges (line 7) and their targets (successors of p). Specifically, for each successor n we evaluate the possible maximum capacity (C_{max}), line 10, that is the minimum between the capacity of its predecessor p ($C_f[p]$) and the frequency of the incoming edge (f_e). Successively, we update the best incoming

[6] First-in first-out.

Algorithm 2: Discover Best Incoming Edges

input : PDFG $\mathcal{G}_p = (T, E_p)$, Source i, Forward Capacities Map C_f, Best Incoming Edges Map E_i

```
1  Create a queue Q;
2  Create a set U ← T ∖ {i};
3  Add i to Q;
4  while Q ≠ ∅ do
5      p ← first node in Q;
6      remove p from Q;
7      for e ∈ p• do
8          n ← target of e;
9          f_e ← frequency of e;
10         C_max ← Min(C_f[p], f_e);
11         if C_max > C_f[n] then
12             C_f[n] ← C_max;
13             E_i[n] ← e;
14             if n ∉ Q ∪ U then add n to U;
15         if n ∈ U then
16             remove n from U;
17             add n to Q;
```

Algorithm 3: Discover Best Outgoing Edges

input : PDFG $\mathcal{G}_p = (T, E_p)$, Sink o, Backward Capacities Map C_b, Best Outgoing Edges Map E_o

```
1  Add o to Q;
2  U ← T ∖ {o};
3  while Q ≠ ∅ do
4      n ← first node in Q;
5      remove n from Q;
6      for e ∈ •n do
7          p ← source of e;
8          f_e ← frequency of e;
9          C_max ← Min(C_b[n], f_e);
10         if C_max > C_b[p] then
11             C_b[p] ← C_max;
12             E_o[p] ← e;
13             if p ∉ Q ∪ U then add p to U;
14         if p ∈ U then
15             remove p from U;
16             add p to Q;
```

edge of n ($E_i[n]$) and its current maximum capacity ($C_f[n]$) if C_{max} is greater than $C_f[n]$, line 11.

Since a change of $C_f[n]$ may entail a change of n successors' maximum capacities, we mark n as unexplored if previously explored (n is added to U, line 14). Finally, if n is unexplored, we add it to the tail of Q (line 17). We then perform the backward exploration starting from o (line 1 to 16). The algorithm concludes retaining the best incoming and outgoing edges, and those with a frequency above the threshold f_{th} (line 21).

Though the third property (i.e. $\mid E_f \mid < 2 \mid T \mid$) can only be guaranteed for $\eta = 1$, the usage of η is meant to balance fitness and precision. Since, the lower is the value of η the more edges may be retained, resulting in a higher fitness at the cost of lower precision and higher control-flow complexity.

Node	C_f	E_i	C_b	E_o
a	∞	-	20	(a,b)
b	60	(a,b)	20	(b,e)
c	20	(a,c)	20	(c,g)
d	20	(a,d)	20	(d,g)
e	40	(b,e)	20	(e,h)
f	20	(b,f)	30	(f,g)
g	20	(f,g)	80	(g,h)
h	20	(g,h)	∞	-

Table 1: Filtering algorithm example.

Figure 2c shows the output of the filtering algorithm when applied to the PDFG previously obtained (Figure 2b). The results of the forward and backward explorations (mappings C_f, E_i, and C_b, E_o) are shown in Table 1. As consequence of retaining the best incoming and outgoing edges for each node, we would drop edges: (e,c) and (c,f). These latter would be removed independently of the value assigned to η.

3.4 Filtered PDFG to BPMN Process Model

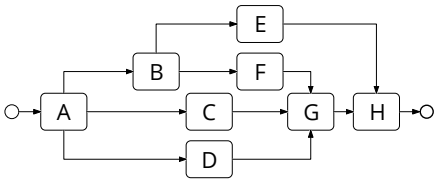
Once completed the processing of the DFG, we can start the conversion from the filtered PDFG to the BPMN process model.

Definition 6 (BPMN process model) A *BPMN process model* (or BPMN model) is a connected graph $\mathcal{M} = (i,o,T,G,E_m)$, where i is the start event, o is the end event, T is a non-empty set of tasks, $G = G^{+} \cup G^{\times} \cup G^{\circ}$ is the union of the set of AND gateways (G^{+}), the set of XOR gateways ($G^{\times}$) and the set of OR gateways (G°), and $E_m \subseteq (T \cup G \cup \{i\}) \times (T \cup G \cup \{o\})$ is the set of edges. Further, given $g \in G$, g is a *split* gateway if it has more than one outgoing edge, i.e. $|\, g\bullet \,| > 1$, or a *join* gateway if it has more than one incoming edge, i.e. $|\, \bullet g \,| > 1$.

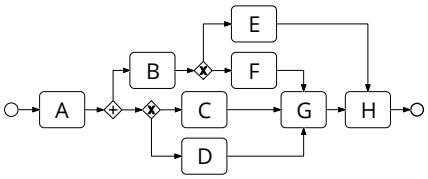
Algorithm 4 highlights the main parts of the conversion. Specifically, we create a start and an end event (lines 1 and 2). Then, we initialize the set of tasks and edges, respectively as the set of nodes of the filtered PDFG, and as the set of the edges of the filtered PDFG plus two new edges: one connecting the start event with the former source of the DFG, and one connecting the former sink of the DFG to the end event (line 6). Lastly, an empty set of gateways is created, which will be filled through the following three steps: split discovery, join discovery and ORs replacement. Figure 3a shows the initialization of the BPMN model obtained from the filtered PDFG of Figure 2c.

3.5 Splits Discovery

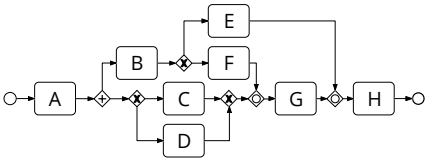
To generate the split gateways, we rely on the concurrency relations identified during the second step of our approach (section 3.2). The splits discovery is based on the idea that tasks directly following (successors of) the same split gateway are concurrent to the same set of tasks which do not directly follow such gateway. With a reference to Figure 4b, being tasks c and d successors of gateway and_1, they are both concurrent to tasks e, f, g, due to gateway and_3 (i.e. $c \| e$, $c \| f$, $c \| g$, and $d \| e$, $d \| f$, $d \| g$). Similarly, being tasks a and b successors of gateway xor_1, they share same empty set



(a) Initial state.



(b) After splits discovery.



(c) After joins discovery.

Fig. 3: Processing of the BPMN model.

Algorithm 4: Filtered PDFG to BPMN process model

input : Filtered PDFG $\mathcal{G}_f = (N, E_f)$
output : BPMN process model $\mathcal{M} = (i, o, G, T, E_m)$

Create a start event i;
Create an end event o;
Create a set $T \leftarrow N$;
$s \leftarrow t \in T \mid |\bullet t| = 0$;
$e \leftarrow t \in T \mid |t\bullet| = 0$;
Create a set $E_m \leftarrow E_f \cup \{(i,s),(e,o)\}$;
Create a set $G \leftarrow \varnothing$;
Create a BPMN process model $\mathcal{M} \leftarrow (i, o, T, G, E_m)$;
DiscoverSplits($\mathcal{M}$);
DiscoverJoins($\mathcal{M}$);
ReplaceORs($\mathcal{M}$);
return $\mathcal{M}$;

Algorithm 5: Discover Splits

input : BPMN process model $\mathcal{M} = (i, o, T, G, E_m)$

for $t \in T$ **do**
  if $|t\bullet| > 1$ **then**
    Create a set $S \leftarrow$ d-successors of t;
    Create a map $C : S \rightarrow 2^S$;
    Create a map $F : S \rightarrow 2^S$;
    for $s_1 \in S$ **do**
      $C[s_1] \leftarrow \{s_1\}$;
      $F[s_1] \leftarrow \varnothing$;
      for $s_2 \in S$ **do**
        if $(s_2 \neq s_1 \wedge s_2 \| s_1)$ **then** add s_2 to $F[s_1]$;
    remove from E_m the outgoing edges of t ;
    while $|S| > 1$ **do**
      discoverXORsplits($\mathcal{M}, S, C, F$);
      discoverANDsplits($\mathcal{M}, S, C, F$);
    $s \leftarrow$ unique element of S;
    add an edge from t to s;

Algorithm 6: Discover XOR-splits

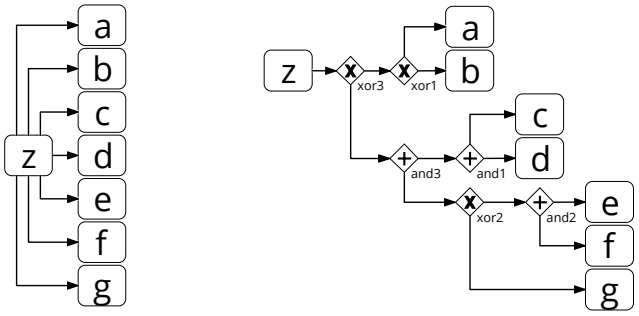
input : BPMN $\mathcal{M}$, Set S, Map C, Map F

do
  Create a set $X \leftarrow \varnothing$;
  for $s_1 \in S$ **do**
    Create a set $C_u \leftarrow C[s_1]$;
    for $s_2 \in S$ **do**
      if $F[s_1] = F[s_2] \wedge s_1 \neq s_2$ **then**
        add s_2 to X;
        $C_u \leftarrow C_u \cup C[s_2]$;
    if $X \neq \varnothing$ **then**
      add s_1 to X;
      break;
  if $X \neq \varnothing$ **then**
    Create an XOR gateway $g_\times$;
    add $g_\times$ to $G^\times$;
    for $s \in X$ **do**
      add an edge from $g_\times$ to s;
      remove s from S;
    add $g_\times$ to S;
    $F[g_\times] \leftarrow F[s_1]$;
    $C[g_\times] \leftarrow C_u$;
while $X \neq \varnothing$;

Algorithm 7: Discover AND-splits

input : BPMN $\mathcal{M}$, Set S, Map C, Map F

do
  Create a set $A \leftarrow \varnothing$;
  for $s_1 \in S$ **do**
    Create a set $C_u \leftarrow C[s_1]$;
    Create a set $F_i \leftarrow F[s_1]$;
    Create a set $CF_{s_1} \leftarrow C[s_1] \cup F[s_1]$;
    for $s_2 \in S$ **do**
      Create a set $CF_{s_2} \leftarrow C[s_2] \cup F[s_2]$;
      if $CF_{s_1} = CF_{s_2} \wedge s_1 \neq s_2$ **then**
        add s_2 to A;
        $C_u \leftarrow C_u \cup C[s_2]$;
        $F_i \leftarrow F_i \cap F[s_2]$;
    if $A \neq \varnothing$ **then**
      add s_1 to A;
      break;
  if $A \neq \varnothing$ **then**
    Create an AND gateway g_+;
    add g_+ to G^+;
    for $s \in A$ **do**
      add an edge from g_+ to s;
      remove s from S;
    add g_+ to S;
    $C[g_+] \leftarrow C_u$;
    $F[g_+] \leftarrow F_i$;
while $A \neq \varnothing$;



(a) Before (b) After

Fig. 4: Splits discovery example.

of concurrent relations, because a, b and all the other tasks are mutually exclusive due to gateway xor_3. Knowing which tasks are successors of the same gateway, we can identify the gateway's type by checking whether its successors are concurrent or mutually exclusive, e.g. c and d are concurrent (i.e. $c \| d$), whilst a and b are mutually exclusive (i.e. $\neg\, a \| b$).

Before explaining in details how we discover a hierarchy of splits, we need to define the following concepts.

Definition 7 (Split-task) Given a BPMN model $\mathcal{M} = (i, o, T, G, E_m)$, a *split-task* is a task $t \in T$, such that $|t\bullet| > 1$.[7]

Definition 8 (P-successor of a Split-task) Given a BPMN model $\mathcal{M} = (i, o, T, G, E_m)$ and a split-task $t \in T$, a *p-successor* of t is a task or gateway $s \in T \cup G$ such that there exists a path from t to s. Further, if $s \in T$ we say s is a *t-successor*, whilst if $s \in G$, s is a *g-successor*.

Definition 9 (D-successor of a Split-task) Given a BPMN model $\mathcal{M} = (i, o, T, G, E_m)$ and a split-task $t \in T$, a *d-successor* of t is a task $s \in T \cup G$ such that there exists an edge from t to s. A d-successor is always a p-successor, but not vice-versa.

Definition 10 (Successor Cover) Given a BPMN model $\mathcal{M} = (i, o, T, G, E_m)$, a split-task $t \in T$ and a p-successor s of t, the *cover* of s is the subset C_s of the t-successors of t such that for each $c \in C_s$ there exists a path from t to c that visits s. The following properties are always true: $s \in C_s$ and $C_s \cap G = \varnothing$.

Definition 11 (Successor Future) Given a BPMN model $\mathcal{M} = (i, o, T, G, E_m)$, a split-task $t \in T$ and a p-successor s, the *future* of s is the subset F_s of the t-successors of t such that $f \in F$ iff $f \| c, \forall\, c \in C_s$.

To describe how Algorithm 5 discovers a hierarchy of splits we use the example in Figure 4, assuming the concurrency relations showed in Figure 4b. Given as input a BPMN model to Algorithm 5, we look for split-tasks (line 1), e.g. z (Figure 4a). We retrieve the d-successors of z (line 3), and for each d-successor, e.g. e, we detect its *cover* and its *future* (maps $C[s]$ and $F[s]$, lines 4 and 5). In our example, the cover and the future of e are the sets $\{e\}$ and $\{c, d, f\}$, respectively (since we assumed $e \| c$,

[7] A *split-task* represents a syntactical error in the BPMN model.

D-successor	C	F
a	a	-
b	b	-
c	c	d, e, f, g
d	d	c, e, f, g
e	e	f, c, d
f	f	e, c, d
g	g	c, d
xor_1	a, b	-
and_1	c, d	e, f, g
e	e	f, c, d
f	f	e, c, d
g	g	c, d
xor_1	a, b	-
and_1	c, d	e, f, g
and_2	e, f	c, d
g	g	c, d
xor_1	a, b	-
and_1	c, d	e, f, g
xor_2	e, f, g	c, d
xor_1	a, b	-
and_3	c, d, e, f, g	-
xor_3	a, b, c, d, e, f, g	-

Table 2: Splits discovery example.

$e \| d$ and $e \| f$). Table 2 shows how the sets $C[s]$ and $F[s]$ evolves during the execution of the algorithm. After computing *cover* and *future* of each d-successor (line 6 to 10), we use them to discover XOR-splits and AND-splits in two phases (lines 13 and 14). In the first phase – Algorithm 6 – we look for all the d-successors sharing the same *future* (line 6). Whenever we find any (line 9), we introduce an XOR-split proceeding these p-successors, which replaces them as d-successor of z. This gateway has as *future* the same shared *future* of the selected d-successors, and as *cover* the union of their *covers*. This is in-line with our initial idea, since d-successors having the same *future* are successors of the same gateway and are not concurrent. Otherwise, if they were concurrent, they would be in each other *futures* and their *futures* would differ.[8] We repeat this operation until no further XOR-splits are identified (line 21). Once all possible XOR-splits are discovered, we move toward the second phase, i.e. the discovery of AND-splits – Algorithm 7.

Unlike the XOR-splits, to identify AND-splits we cannot rely only on the d-successors' *futures*. Instead, we select the d-successors having the same set of nodes resulting from the union of their *cover* and *future* (line 9 to 15). Then, an AND-split is introduced before these d-successors. This AND-split has as *future* the intersection of the *futures* of the set of the selected d-successors and as *cover* the union of their *covers*. Likewise for the XOR-split, the AND-split becomes a new d-successor replacing the p-successors that will now belong to its cover. The discovery of the AND-splits is still in-line with our initial idea. Indeed, given a set of candidate AND-split successors (i.e. d-successors having a common subset of their *futures*), and removing the common subset from their *futures*, the genuine AND-split successors will be those for which their *cover* is contained in the *future* of each other candidate successor of the AND-split (i.e. successors of the same AND-split must be concurrent each others). In our example (Table 2), c and d have as shared future the subset $F_s = \{e, f, g\}$. If we remove

[8] This happens because by construction a task cannot belong to its own future.

this subset, we would see that the cover of c matches the future of d and vice-versa (i.e. $c \| d$ and $d \| c$). The fact that we look for d-successors having the same union of their *cover* and *future* is a mere computational optimization, since by construction the intersection of the covers of two different d-successors is always empty, and the intersection between the cover and the future of a d-successor is always empty.

We repeat Algorithms 6 and 7 until the split-task becomes a normal task, having just one d-successor (Algorithm 5, line 12). Figure 3b shows the output of this step for our working example, when we give as input to Algorithm 5 the BPMN models showed in Figure 3a.

3.6 Joins Discovery

Once all the split gateways have been placed, we can discover the join gateways. To do so, we rely on the Refined Process Structure Tree (RPST) [28] of the current BPMN model. The RPST of a process model is a tree where its nodes represent the single-entry single-exit (SESE) fragments of the process model and its edges denote a containment relation between SESE fragments. Specifically, the children of a SESE fragment are its directly contained SESE fragments, whilst SESE fragments on different branches of the tree are disjoint. Since each SESE fragment is a subgraph of the process model, and the partition of the process model into SESE fragments is made in terms of edges, a single node (of the process model) can be shared by multiple SESE fragments. Further, each SESE fragment can be of one of the four types: a *trivial* fragment consists of a single edge; a *polygon* is a sequence of fragments; a *bond* is a fragment where all the children fragments share two common nodes, one being the entry and the other being the exit of the bond; any other fragment is a *rigid*. Lastly, each SESE can be classified as *homogeneous* if the gateways it contains (and are not contained in any of its SESE children) are all of the same type (e.g. only XOR-gateways), or *heterogeneous* if such gateways have different types.

To explain Algorithm 8, we need to introduce the concept of *loop-join*.

Definition 12 (Loop-edge and Loop-joins) Given a BPMN model $\mathcal{M} = (i, o, T, G, E_m)$, and an edge $e = (a, b) \in E_m$, e is a *loop-edge* iff a is a node topologically deeper than b and there exists a path from b to a. Further, if $|\bullet b| > 1$, we refer to b as *loop-join*.

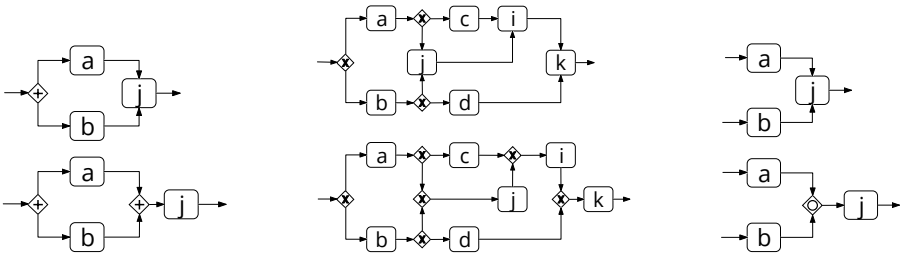
The first step of Algorithm 8 is to generate the RPST of the input BPMN model. Then, we add all the RPST nodes to a queue (Q) ordering the nodes bottom-up, i.e. leaves to root (line 2), and we analyse each of these nodes (which are the SESE fragments composing the BPMN model). Precisely, for each task (t) having multiple incoming edges within the SESE fragment, we create a new join gateway (g) and we redirect all the incoming edges of t to g, line 12. Finally, we set the type of g according to the following rules: if g is a loop-join, it is turned into a XOR; else if t is within a homogeneous SESE fragment we match the type of the homogeneous SESE fragment, otherwise the type is set to OR. These rules guarantee soundness for acyclic models and deadlock-freedom for cyclic models as discussed below.

Figure 5 shows how our approach works for bonds (Fig. 5a), for homogeneous rigids (Fig. 5b), and for all other cases, i.e. heterogeneous rigid (Fig. 5c).

Considering the working example in Fig. 3b, we detect three joins. The first one is the XOR-join at the exit of the bond containing tasks c, d and g. Being the entry of this bond an XOR-split, the bond is XOR-homogeneous, so that the type of the

Algorithm 8: Discover Joins

```
input : BPMN process model M = (i, o, T, G, E_m)
generate the RPST from M;
Create a queue Q ← nodes of the RPST bottom-up ordered;
while Q ≠ ∅ do
    n ← first node in Q;
    remove n from Q;
    Create a set T_n ← tasks composing n;
    Create a set E_n ← edges composing n;
    for t ∈ T_n do
        if | • t ∩ E_n |> 1 then
            create a gateway g;
            add an edge from g to t;
            for e ∈ • t ∩ E_n do set target of e to g;
            if •g contains backedges then
                set the type of g to XOR;
            else
                if n is homogeneous then
                    set the type of g equal to the type of n;
                else
                    set the type of g to OR;
```



(a) Bond. (b) Homogeneous Rigid. (c) Generic case.

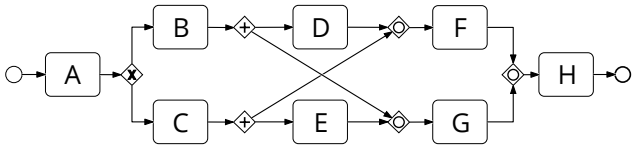
Fig. 5: Joins discovery examples.

joins is set to XOR. The remaining two joins are within the parent SESE fragment of the bond, which is a heterogeneous rigid, hence, we use two OR-joins. The resulting model is shown in Fig. 3c.

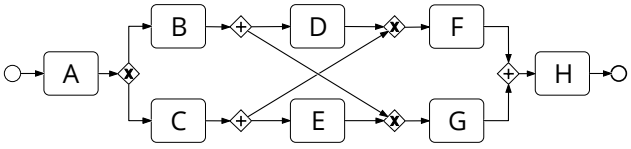
3.7 OR-joins Minimization

The approach described in Section 3.6 avoids the placement of trivial OR-joins within bonds and homogeneous rigids, but it does not prevent an abuse of OR-joins in case of heterogeneous rigids. Since we aim to be compliant with the 7PMG [22] (Section 2), according to the fifth guideline, we should avoid the use of OR gateways where possible. To achieve this goal, we designed an algorithm able to minimize the use of the OR-joins opportunely replacing the trivial ones[9] with XOR or AND joins. Since the algorithm is centred on the concept of *minimal dominator*, we introduce it formally.

[9] An OR-join is said trivial when its semantic is equivalent to the semantic of an XOR or AND join.



(a) Before.



(b) After.

Fig. 6: OR-joins minimization example.

Definition 13 (Minimal Dominator) Given a BPMN model $\mathcal{M} = (i, o, T, G, E_m)$, an OR-join $j_\circ \in G^\circ$, and a split gateway $d \in G$, d is a *dominator* of $j_\circ$ iff all the paths from i to $j_\circ$ visit d. Furthermore, d is the *minimal dominator* of $j_\circ$ iff none of the paths from d to $j_\circ$ visits other dominators of $j_\circ$.

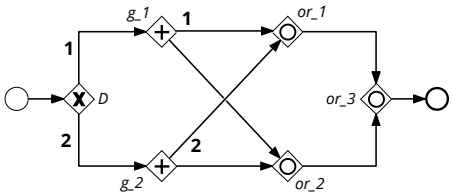
Algorithm 9: Check OR-join Semantic

input : A BPMN model $\mathcal{M} = (i, o, T, G, E_m)$, Gateway $j_\circ \in G^\circ$
output : Semantic of $j_\circ$

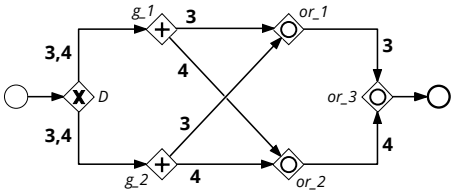
$d \leftarrow$ minimal dominator of $j_\circ$;
Create a set $G_s \leftarrow$ split gateways in any path from d to $j_\circ$;
Create a set $E_s \leftarrow$ outgoing edges of all the gateways in G_s;
Create a map $T : E_s \rightarrow 2^{\bullet j_\circ}$;
semantic $\leftarrow$ null;
for $e \equiv (g_s, x) \in E_s$ **do**
    $T[e] \leftarrow \{e_{ij} \in \bullet j_\circ \mid \exists$ a path from x to $e_{ij}\}$;
    if $T[e] = \varnothing \wedge$ *semanticOf*(g_s) = *XOR* **then** semantic = XOR;
for $g \in G_s$ **do**
    for $e_1 \in g\bullet$ **do**
        for $e_2 \in g\bullet$ **do**
            $I \leftarrow T[e_1] \cap T[e_2]$;
            $S_1 \leftarrow T[e_1] \setminus I$;
            $S_2 \leftarrow T[e_2] \setminus I$;
            if $S_1 \neq \varnothing \wedge S_2 \neq \varnothing$ **then**
                if *semantic* ! = *null* $\wedge$ *semantic* ! = *semanticOf(g)* **then**
                    return OR;
                else
                    semantic $\leftarrow$ semanticOf(g);
return semantic;

The procedure for the *OR-joins minimization* detects the trivial OR-joins of the input BPMN model, and replaces them with XOR or AND joins according to their semantics (proof in Section 4). Figure 6 shows an example of the OR-joins minimization. We describe the idea behind this procedure before presenting the algorithm.

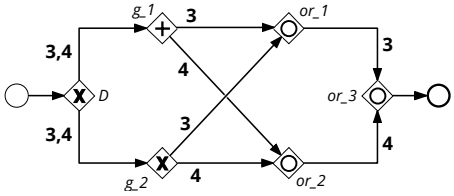
Given a BPMN model $\mathcal{M} = (i, o, T, G, E_m)$, an OR-join $j_\circ$, and its minimal dominator $d \in G$, we know that all tokens that may arrive to one or more incoming edge of $j_\circ$ (*incoming tokens*) are generated by d.



(a) Replacing of OR_1 and OR_2.



(b) Replacing of OR_3.



(c) Non-trivial OR_3.

Fig. 7: OR-joins minimization algorithm.

By checking the semantic of all the split gateways visited by the incoming tokens in all the paths from d to $j_\circ$, we identify how the incoming tokens split (*split relations*), i.e. mutually exclusive or concurrently. Finally, if the split relations between the incoming tokens of $j_\circ$ are all the same, $j_\circ$ is a trivial OR-join, and it can be replaced with an XOR-join if the split relations are of mutually exclusive type or an AND-join if concurrent. Otherwise, the OR-join is not trivial, and its replacement must be handled differently. Algorithm 9 shows in details how we check the semantic of an OR-join, to decide if we can replace it. Given a BPMN model $\mathcal{M} = (i, o, T, G, E_m)$ and an OR-join $j_\circ$, we retrieve the minimal dominator of $j_\circ$ (d, line 1). We create a set (G_s) containing all the split gateways visited on the paths from d to $j_\circ$; and a set (E_s) containing all the outgoing edges of the split gateways in G_s. Then, for each edge $e \equiv (g_s, x) \in E_s$, we initialize a set ($T[e]$) containing the edges $e_{ij} \in \bullet j_\circ$ such that there exists a path from x to e_{ij} (i.e. $T[e]$ contains the incoming edges of $j_\circ$ that may receive a token from e). During this initialization, we may find an empty set $T[e]$. In such case, if g_s (source of e) is an XOR-split, the empty set $T[e]$ indicates that an incoming token of $j_\circ$ may escape on g_s from e. Therefore, since there exists a case where $j_\circ$ does not receive a token on one of its incoming edges, $j_\circ$ cannot be a trivial AND-join. We record this occurrence setting the possible trivial semantic of the OR-join as exclusive (XOR, line 8). Initialized the sets $T[e]$, we can check the split relations between the incoming

tokens of $j_\circ$. Precisely, for each split gateway $g \in G_s$, and for each pair of its outgoing edges (e_1 and e_2, lines 9 to 11), we compute the intersection of $T[e_1]$ and $T[e_2]$. This intersection (I) contains the edges of $\bullet j_\circ$ that may receive incoming tokens either from e_1 or e_2 (i.e. incoming tokens for the edges in I do not split between e_1 and e_2). By removing I from $T[e_1]$ and $T[e_2]$ (sets S_1 and S_2, lines 13 and 14), we detect the incoming tokens that split between e_1 and e_2, i.e. S_1 (S_2) contains the incoming edges of $j_\circ$ that cannot receive tokens from e_2 (e_1). If S_1 and S_2 are both not empty, we identified a split relation between the incoming tokens of the edges in S_1 and the incoming tokens of the edges in S_2. Consequently, the incoming tokens that split between e_1 and e_2 may arrive to different incoming edges of $j_\circ$ with the same semantic of g_s (i.e. the gateway where they split). Therefore, $j_\circ$ is trivial only if all the incoming tokens match that semantic (line 16). Figure 7, shows graphically how Algorithm 9 works on each OR-join of the model in Figure 6a. Specifically, Figure 7a highlights where the incoming tokens of the OR-join OR_1 split. The minimal dominator of OR_1 (as well for OR_2 and OR_3) is the XOR-gateway D. By running Algorithm 9 on OR_1, we detect that the two incoming tokens (namely 1 and 2) split only on D. Being the semantic of D exclusive, we can replace OR_1 with an XOR-join. The semantic check for OR_2 is equivalent to the one of OR_1. For OR_3 (Figure 7b), its incoming tokens (namely 3 and 4) may split on g_1 and g_2, but they do not split on D. Since g_1 and g_2 are both AND-splits, we can replace OR_3 with and AND-split. In such example, all three OR-joins were trivial. Differently, if g_2 is an XOR-split (figure 7c), OR_1 and OR_2 would still be replaced with XOR-joins, whilst OR_3 would remain an OR-join. This would happen because the incoming tokens of OR_3 split on g_2 (now XOR) and on g_1 (still AND), meaning that their semantics may be either exclusive or concurrent, according to where they would split during the execution of the process model (either on g_1 or g_2).

3.8 Complexity

Let n be the number of events in the log and m be the number of tasks (distinct nodes of the DFG). The DFG construction is in $O(n)$, since we sequentially read each event and generate the respective node in the graph, simultaneously incrementing the directly-follows and short-loop frequencies. The self-loops discovery is linear on the number of nodes of the DFG, hence in $O(m)$. The short-loops discovery is done on pairs of tasks, so this step is performed in $O(m^2)$. The filtering complexity is dominated by the forward (backward) exploration. It explores each node a number of times equal at most to the number of edges of the graph, and for each node exploration it loops on the outgoing (incoming) edges. In the worst scenario, the maximum number of edges is equal to m^2 (e.g. an edge for each pair of nodes), consequently, the filtering is in $O(m^4)$. The split discovery is in $O(m^4)$, because we may run Algorithm 5 for each node, which executes m times Algorithm 6 and 7, and these latter have two nested loops on m. The join discovery complexity is dominated by the three nested loops: the one on the number of nodes of the RPST, the one on the number of tasks, and the one on the number of edges. Since the RPST contains a number of nodes equal at most to the number of edges of the model, the join discovery is in $O(m^5)$. For the OR-minimization, we run Algorithm 9 for each OR-join – m times, i.e. one join for each task. The complexity of Algorithm 9 is dominated by its three nested loops. The outer loop is on the number of split gateways – bound by m, i.e. one split for each task –, whilst the two inner loops are on the number of edges. Therefore, the OR-minimization is in $O(m^6)$. We can conclude that Split Miner is in $O(n + m^6)$.

4 Semantic Properties of the Discovered Model

In this section, we provide formal proofs of some semantic properties of the BPMN process model discovered by Split Miner. Precisely, we show that in the case of *acyclic* BPMN process models, Split Miner guarantees soundness and the absence of any trivial OR-joins. Moreover, for *cyclic* BPMN process models, it is not possible to guarantee the soundness, but only deadlock-freedom. This latter result is ensured by the semantic of the OR-joins [34].

In the following, we refer to the BPMN process model as workflow graph.

Definition 4.1 (Workflow graphs)
A *workflow graph* is a triple $G := (V, E, l)$, where (V, E) is a finite directed graph consisting of a set V of nodes and a set $E \subseteq V \times V$ of edges, and $l : V \to \{AND, XOR, OR\}$ is a partial mapping such that:

1. there is exactly one *source* node, where a node $v \in V$ is a *source* if and only if it has exactly one outgoing edge and no incoming edges,
2. there is at least one *sink* node, where a node $v \in V$ is a *sink* if and only if it has exactly one incoming edge and no outgoing edges,
3. If $l(v)$, $v \in V$, is defined, then v is neither the source nor a sink,
4. If $v \in V$ is a gateway, then $l(v)$ is defined, and
5. every node $v \in V$ is on a directed path from the source to some sink.

We chose this formalism to ease the readability of the proofs and their symbolism. It is important to notice that by definition a workflow graph $G := (V, E, l)$ is equivalent to a BPMN process model $\mathcal{M} = (i, o, T, G, E_m)$. Where $\{i\} \cup \{o\} \cup T \cup G \equiv V$, $E_m \equiv E$, i and o are the only source and sink of the workflow graph, and the type of the gateways is identified by the function l.

4.1 Preliminaries

Let $G := (V, E, l)$ be a workflow graph. A *state* of G is a mapping $s : E \to \mathbb{N}_0$.[10]

Definition 4.2 (Semantics of workflow graphs)
A *state transition* of a workflow graph $G := (V, E, l)$ is a triple (s_1, v, s_2), also written as $s_1[v\rangle s_2$, where s_1 and s_2 are states of G, $v \in V$, and one of these three conditions holds:

1. $l(v) \notin \{XOR, OR\}$ and
$$s_2(e) = \begin{cases} s_1(e) + 1 & e \in E \text{ is an outgoing edge of } v \\ s_1(e) - 1 & e \in E \text{ is an incoming edge of } v \\ s_1(e) & \text{otherwise.} \end{cases}$$
2. $l(v) = XOR$ and there exists an incoming edge $e_1 \in E$ of v and an outgoing edge $e_2 \in E$ of v such that:
$$s_2(e) = \begin{cases} s_1(e) + 1 & e \in E \text{ and } e = e_2 \\ s_1(e) - 1 & e \in E \text{ and } e = e_1 \\ s_1(e) & \text{otherwise.} \end{cases}$$
3. $l(v) = OR$ for each edge $e' \in E$ and each incoming edge $e \in E$ of v such that $s_1(e') \geq 1$ and $s_1(e) = 0$ there is no directed path from e' to e, and there exists a

[10] By $\mathbb{N}_0$, we denote the set of all natural numbers including zero.

nonempty set F of outgoing edges of v such that:

$$s_2(e) = \begin{cases} s_1(e) + 1 & e \in F \\ s_1(e) - 1 & e \in E \text{ is an incoming edge of } v \text{ such that } s_1(e) \geq 1 \\ s_1(e) & \text{otherwise.} \end{cases}$$

Let e be the only outgoing edge of the source of a workflow graph $G := (V, E, l)$. Then, state s of G for which it holds that $s(e) = 1$ and for every $e' \in E$ such that $e' \neq e$ it holds that $s(e') = 0$ is the *initial state* of G. Let s_0 be the initial state of G. A state s is a *reachable state* of G if and only if $s = s_0$ or there is a sequence of nodes $\sigma := \langle v_1, \ldots, v_n \rangle \in V^*$, $n \in \mathbb{N}$, such that for every position i of σ it holds that $s_{i-1}[v_i\rangle s_i$ is a state transition of G, and $s = s_n$. A state s is a *final state* of G if and only if for every $v \in V$ there exists no state s' of G such that (s, v, s') is a state transition of G. A final state s of G is a *deadlock* if and only if there is an edge $e \in E$ such that $s(e) > 0$ and e is not an incoming edge of some sink of G. A state s of G is *safe* if and only if for all $e \in E$ it holds that $s(e) \leq 1$; otherwise s is *unsafe*. A workflow graph $G := (V, E, l)$ is *safe* if and only if all its reachable states are safe; otherwise G is *unsafe*.

Definition 4.3 (Sound worfklow graphs)
A workflow graph G is *sound* if and only if G is safe and has no reachable deadlocks.

Given a node $v \in V$ of a workflow graph $G := (V, E, l)$, by $\bullet v$ we denote the set of all incoming edges of v. Whereas by $v\bullet$, we denote the set of all outgoing edges of v. In what follows, without loss of generality, we assume that for every vertex v of every workflow graphs it does not hold that $|\bullet v| > 1$ and $|v \bullet| > 1$. A node $v \in V$ is a *gateway* if and only if it holds that $|\bullet v| > 1$ or $|v \bullet| > 1$. A gateway $v \in V$ is a *split* if and only if $|\bullet v| > 1$. A gateway $v \in V$ is a *join* if and only if $|v \bullet| > 1$. Hence, every gateway is either a split or a join, but not both.

Let $G := (V, E, l)$ be a triple, where (V, E) is a finite directed graph consisting of a set V of nodes and a set $E \subseteq V \times V$ of edges, and $l : V \to \{AND, XOR, OR\}$ is a partial mapping. A *prefix* of G is a triple $G' := (V', E', l')$, denoted by $G' \sqsubseteq G$, where $V' \subseteq V$ is such that if $v \in V'$ then for every $v' \in V$ for which $(v', v) \in E^+$ it holds that $v' \in V'$, $E' \subseteq E$ is such that (V', E') is a connected graph, and $l' := l|_{dom(l) \cap V'}$. We write $G' \sqsubset G$ if and only if $G' \sqsubseteq G$ and $G' \neq G$.

Let $G := (V, E, l)$ be a prefix of a workflow graph and let $X \subseteq V$ be all the nodes of G such that for every $x \in X$ it holds that $x\bullet = \emptyset$. A *completion* of G is a triple $G' := (V', E', l)$, where $V' := V \cup Y$, $Y \cap V = \emptyset$, $|X| = |Y|$, and there is a bijection b from X to Y, and $E' := E \cup b$.

4.2 Proofs

It is easy to see that a completion of a workflow graph is again a workflow graph.

Corollary 4.4 (Completion is a workflow graph)
A completion of a prefix of a workflow graph is a workflow graph.

Corollary 4.4 follows immediately from its definition.

Split Miner produces models with OR-joins and then applies Algorithm 9 to replace trivial OR-joins with AND- and XOR-joins. Next, we demonstrate that an acyclic workflow graph in which all joins are OR-joins is guaranteed to be sound.

Lemma 4.5 (Sound acycic workflow graphs)
If a workflow graph $G := (V,E,l)$ is acyclic, i.e., E^+ is irreflexive, such that for every split $v \in V$ it holds that $l(v) \neq OR$ and for every join $v \in V$ it holds that $l(v) = OR$, then G is sound.

Proof (Sketch) By Noetherian induction on prefixes of G, we show that a completion of G is sound. Let $\mathcal{G}$ be the set of all prefixes of G. Then, $(\mathcal{G}, \sqsubseteq)$ is a well-founded set.

Induction basis: A completion of $(s,\varnothing,\varnothing)$, where $s \in V$ is the source of G, is sound. Clearly, a workflow graph $(\{s,x\},\{(s,x)\},\varnothing)$, where $s \neq x$, is sound.

Induction step: Let $G' := (V',E',l') \in \mathcal{G}$ be a prefix of G. Assume that for every $G'' \in \mathcal{G}$ such that $G'' \sqsubset G'$ it holds that a completion of G'' is sound. Let $\hat{G} := (\hat{V},\hat{E},\hat{l}) \sqsubset G'$ be such that $E' \setminus \hat{E} = \{e\}$, $e \in E$. We distinguish these two cases:

1. The target node v of e is in $\hat{V}$. Then, v is a join for which it holds that $l(v) = OR$; indeed, for every join $j \in V$ it holds that $l(j) = OR$. By definition of the semantics of workflow graphs, refer to Definition 4.2, and because G and, thus, G' are acyclic, it holds that a completion of G' is sound. The only interesting case here is when for the source v' of e it holds that $l(v') = XOR$ and $|v' \bullet| > 1$. In this case, one cannot reach a deadlock or unsafe state from a state s of a completion of G' for which $s(e) = 1$ because for every join j that can be reached from v' via a directed path it holds that $l(j) = OR$.
2. The target node v of e is not in $\hat{V}$. Then, v is not a join and, clearly, a completion of G' is sound. Note that a deadlock or unsafe reachable marking in a workflow graph can be introduced only via a fresh join.

Hence, a completion of G is sound. It is easy to see that if a completion of G is sound, then G is also sound. ∎

Before showing that a replacement of a trivial OR-join preserves soundness of an acyclic workflow graph, we define this transformation formally.

Definition 4.6 (Replacement of OR join)
Let $G := (V,E,l)$ be a workflow graph, such that for every split $v \in V$ it holds that $l(v) \neq OR$. Let $v \in V$ be a join, such that $l(v) = OR$. Let s be the result of Algorithm 9 for the input of G and v. The result of *replacement* of v in G is a workflow graph $G' := (V,E,l')$, where $l' := \{(x,y) \in l \mid x \neq v\} \cup \{(v,s)\}$

Next, we demonstrate that replacement of an OR-join in a sound acyclic workflow graph results in a sound workflow graph.

Lemma 4.7 (Replacement of OR join)
Let $G := (V,E,l)$ be a sound acyclic workflow graph such that for every split $v \in V$ it holds that $l(v) \neq OR$. Let $v \in V$ be a join, such that $l(v) = OR$. If $G' := (V,E,l')$ is the result of replacement of v in G, then G' is sound.

Proof (Sketch) By definition of Algorithm 9. We distinguish three cases:

- $l'(v) = OR$. It holds that $G' = G$ and, thus G' is sound.
- $l'(v) = AND$. According to Algorithm 9, it holds that for all the splits on all the paths from the minimal dominator d of v to v are AND-splits. Also, it holds that no two distinct outgoing edges of a split on a path from v to d lead to the same incoming edge of v. Hence, it holds that from every reachable state that marks an incoming edge of v one can reach a state that marks all the incoming edges of v. Hence, the sets of all the reachable states of G and G' are the same. Thus, G' is sound.

- $l'(v) = XOR$. According to Algorithm 9, it holds that for all the splits on all the paths from the minimal dominator d of v to v are XOR-splits. Also, it holds that no two distinct outgoing edges of a split on a path from v to d lead to the same incoming edge of v. Hence, it holds that from every reachable state that marks exactly one incoming edge of v one cannot reach a state that marks two incoming edges of v. Hence, the sets of all the reachable states of G and G' are the same. Thus, G' is sound.

■

Clearly, one can apply replacements to all the OR-joins to obtain a sound acyclic workflow graph without trivial OR-joins.

Theorem 4.8 (Replacement of OR joins)
Let $G := (V, E, l)$ be a sound acyclic workflow graph such that for every split $v \in V$ it holds that $l(v) \neq OR$. Let $f : V' \to \{XOR, AND, OR\}$, where V' is the set of all joins v of G for which it holds that $l(v) = OR$, and $f(v)$, $v \in V'$, is the result of Algorithm 9 for the input of G and v. If $G' := (V, E, l')$, where $l' := \{(x, y) \in l \mid y \neq OR\} \cup \{(x, f(x)) \in V' \times \{XOR, AND, OR\} \mid x \in V'\}$, then G' is sound. ⌟

The proof of Theorem 4.8 follows immediately from Lemma 4.7 and the observation that the order of replacements of OR joins in a sound acyclic workflow graph does not influence the result. The latter fact holds (i) because G and G' have the same structure, i.e., the same nodes and edges, and (ii) because replacements of OR joins preserve the semantics of splits, i.e., for every $v \in V$ such that $|v\bullet| > 1$ it holds that $l(v) = l'(v)$; note that the result of Algorithm 9 depends only on these two factors.

5 Evaluation

We implemented Split Miner (hereafter SM) as a standalone Java application.[11] 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.

5.1 Datasets

We used the collection of real-life event logs provided by the 4TU Centre for Research Data as of August 2017.[12] These logs include all the logs of the annual *Business Process Intelligence Challenge* (BPIC), 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 [10] 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

[11] Available at `http://apromore.org/platform/tools`
[12] `https://data.4tu.nl/repository/collection:event_logs_real`

to the complexity of these logs. Table 3 reports the statistics of the event logs (after the initial filtering where applicable). The dataset is heterogeneous in the number of traces (681 to 150,370), in the number of distinct traces (183 to 8767), in the number of event classes (7 to 82), and in the trace length (1 to 185 events).

Log Name	Total Traces	Distinct Traces	Total Events	Distinct Events	Trace Length		
					min	avg	max
BPIC12	13087	4366	262200	36	3	20	175
$BPIC13_{cp}$	1487	183	6660	7	1	4	35
$BPIC13_{inc}$	7554	1511	65533	13	1	9	123
$BPIC14_{f}$	41353	14948	369485	9	3	9	167
$BPIC15_{1f}$	902	295	21656	70	5	24	50
$BPIC15_{2f}$	681	420	24678	82	4	36	63
$BPIC15_{3f}$	1369	826	43786	62	4	32	54
$BPIC15_{4f}$	860	451	29403	65	5	34	54
$BPIC15_{5f}$	975	446	30030	74	4	31	61
$BPIC17_{f}$	21861	8767	714198	41	11	33	113
RTFMP	150370	231	561470	11	2	4	20
SEPSIS	1050	846	15214	16	3	14	185

Table 3: Statistics of the event logs employed.

5.2 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 ($S\text{-}HM_6$), and Fodina Miner (FO).

Since SM takes as input two thresholds, namely 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.[13] IM takes as input only one threshold for noise filtering. SM, HM_6 (as well $S\text{-}HM_6$) 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, $S\text{-}HM_6$ and FO. For the default parameters evaluation, we measured the quality of the produced models using all the quality dimensions discussed in Section 2.1, 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 2). Further, we report about the average execution time of 5 executions holding out the first (to reduce data loading time bias). 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 [9]. This conversion was

[13] We did not hyperparameter-optimize ETM due to the prohibitively high execution times of this method.

possible for all event logs employed because none of the process models produced by split miner on these logs contained any OR-join.

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.

All the tools required to reproduce the experiments and all the experimental results are available at `https://doi.org/10.6084/m9.figshare.5684119.v1` .

5.3 Results

Table 4 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).

Figure 8 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 input parameters) 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 4 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 across the whole dataset. 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 except for the log $BPIC13_{cp}$, while ETM achieves the highest precision in about half of logs. The plots of Fig. 8 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 across the whole dataset. 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 often high: over 50% in 7 logs, 4 of which fully structured. 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 (one hour). In contrast, SM discovered all the process models in less than one second (on average), being always 2-16 times faster than the second fastest method on every log.

As an example, Figs. 9 and 10 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

Log Name	Discovery Method	Accuracy			Gen. (3-Fold)	Complexity			Sound?	AVG Exec. Time (sec)
		Fitness	Precision	F-score		Size	CFC	Struct.		
BPIC12	SM	0.96	0.81	**0.88**	**0.88**	**51**	41	0.67	yes	**0.22**
	IM	**0.98**	0.50	0.66	0.66	59	37	**1.00**	yes	5.64
	ETM	0.33	**0.98**	0.49	-	69	**10**	**1.00**	yes	3,600
	HM_6	-	-	-	-	89	197	0.28	no	1.78
	S-HM_6	-	-	-	-	86	46	0.20	no	400.78
	FO	-	-	-	-	102	117	0.13	no	13.33
$BPIC13_{cp}$	SM	**0.99**	0.94	**0.96**	**0.96**	11	6	**1.00**	yes	**0.01**
	IM	0.82	**1.00**	0.90	0.90	**9**	**4**	**1.00**	yes	0.06
	ETM	**0.99**	0.76	0.86	-	11	17	**1.00**	yes	3,600
	HM_6	-	-	-	-	13	8	-	no	0.07
	S-HM_6	0.94	0.93	0.94	0.95	13	**4**	**1.00**	yes	0.09
	FO	-	-	-	-	25	23	0.60	no	0.07
$BPIC13_{inc}$	SM	**0.98**	0.92	**0.95**	**0.95**	**12**	8	**1.00**	yes	**0.03**
	IM	0.92	0.50	0.65	0.68	13	**7**	**1.00**	yes	0.62
	ETM	0.84	0.80	0.82	-	28	24	**1.00**	yes	3,600
	HM_6	0.91	0.96	0.93	0.93	13	9	**1.00**	yes	0.53
	S-HM_6	0.91	**0.98**	0.94	0.94	13	9	**1.00**	yes	0.60
	FO	-	-	-	-	44	55	0.75	no	1.48
$BPIC14_f$	SM	0.77	0.93	**0.84**	**0.85**	**20**	**14**	**1.00**	yes	**0.20**
	IM	**0.89**	0.71	0.79	0.79	31	18	**1.00**	yes	2.57
	ETM	0.68	**0.94**	0.79	-	22	15	**1.00**	yes	3,600
	HM_6	-	-	-	-	44	56	-	no	2.59
	S-HM_6	-	-	-	-	120	51	0.27	no	20.72
	FO	-	-	-	-	37	46	0.41	no	37.81
$BPIC15_{1f}$	SM	0.90	**0.90**	**0.90**	**0.89**	111	45	0.51	yes	**0.15**
	IM	0.97	0.57	0.71	0.72	164	108	**1.00**	yes	0.39
	ETM	0.57	0.89	0.69	-	**73**	**21**	**1.00**	yes	3,600
	HM_6	-	-	-	-	150	98	-	no	0.43
	S-HM_6	-	-	-	-	228	122	0.57	no	152.46
	FO	**1.00**	0.76	0.87	0.86	146	91	0.26	yes	0.96
$BPIC15_{2f}$	SM	0.77	**0.91**	**0.83**	**0.83**	126	45	0.31	yes	0.10
	IM	0.93	0.56	0.70	0.70	193	123	**1.00**	yes	0.54
	ETM	0.62	0.90	0.73	-	**78**	**19**	**1.00**	yes	3,600
	HM_6	-	-	-	-	194	158	0.11	no	0.41
	S-HM_6	**0.98**	0.60	0.75	0.76	265	163	0.34	yes	199.57
	FO	-	-	-	-	195	159	0.09	no	2.03
$BPIC15_{3f}$	SM	0.79	**0.93**	**0.85**	**0.85**	94	33	0.48	yes	**0.09**
	IM	**0.95**	0.55	0.70	0.69	159	108	**1.00**	yes	0.83
	ETM	0.66	0.88	0.75	-	**78**	**26**	**1.00**	yes	3,600
	HM_6	**0.95**	0.67	0.79	0.79	157	151	0.07	yes	0.57
	S-HM_6	**0.95**	0.61	0.74	0.75	215	183	0.30	yes	154.19
	FO	-	-	-	-	174	164	0.06	no	1.72
$BPIC15_{4f}$	SM	0.73	0.90	**0.81**	**0.82**	100	35	0.24	yes	**0.06**
	IM	0.96	0.58	0.73	0.72	162	111	**1.00**	yes	0.48
	ETM	0.66	**0.95**	0.78	-	**74**	**17**	**1.00**	yes	3,600
	HM_6	-	-	-	-	158	129	0.15	no	0.39
	S-HM_6	**0.99**	0.65	0.78	0.78	207	137	0.29	yes	143.26
	FO	-	-	-	-	157	127	0.15	no	1.11
$BPIC15_{5f}$	SM	0.79	**0.94**	**0.86**	**0.85**	106	34	0.28	yes	**0.09**
	IM	0.94	0.18	0.30	0.61	134	95	**1.00**	yes	0.40
	ETM	0.58	0.89	0.70	-	**82**	**26**	**1.00**	yes	3,600
	HM_6	-	-	-	-	166	124	0.15	no	0.42
	S-HM_6	**1.00**	0.68	0.81	0.81	239	151	0.43	yes	159.47
	FO	**1.00**	0.71	0.83	0.83	166	125	0.15	yes	1.30
$BPIC17_f$	SM	0.96	0.85	**0.90**	**0.90**	**31**	18	**1.00**	yes	**1.00**
	IM	**0.98**	0.70	0.82	0.82	35	20	**1.00**	yes	6.23
	ETM	0.72	**1.00**	0.84	-	**31**	**5**	**1.00**	yes	3,600
	HM_6	-	-	-	-	36	18	-	no	5.68
	S-HM_6	0.95	0.52	0.67	0.74	42	13	**1.00**	yes	5.74
	FO	-	-	-	-	98	82	0.25	no	55.20
RTFMP	SM	**1.00**	**0.97**	**0.98**	**0.98**	**22**	17	0.46	yes	**0.20**
	IM	0.99	0.63	0.77	0.80	34	20	**1.00**	yes	5.17
	ETM	0.79	0.98	0.87	-	46	33	**1.00**	yes	3,600
	HM_6	-	-	-	-	47	51	0.13	no	4.80
	S-HM_6	0.98	0.95	0.96	0.96	163	97	**1.00**	yes	274.35
	FO	**1.00**	0.94	0.97	0.97	31	32	0.19	yes	2.57
SEPSIS	SM	0.76	**0.86**	**0.81**	**0.81**	33	23	0.91	yes	**0.02**
	IM	**0.99**	0.48	0.65	0.61	50	32	**1.00**	yes	0.23
	ETM	0.71	0.84	0.77	-	**30**	**15**	**1.00**	yes	3,600
	HM_6	-	-	-	-	82	137	0.17	no	0.12
	S-HM_6	0.92	0.42	0.58	0.58	225	131	**1.00**	yes	255.63
	FO	-	-	-	-	60	63	0.28	no	0.23

Table 4: Default parameters evaluation results.

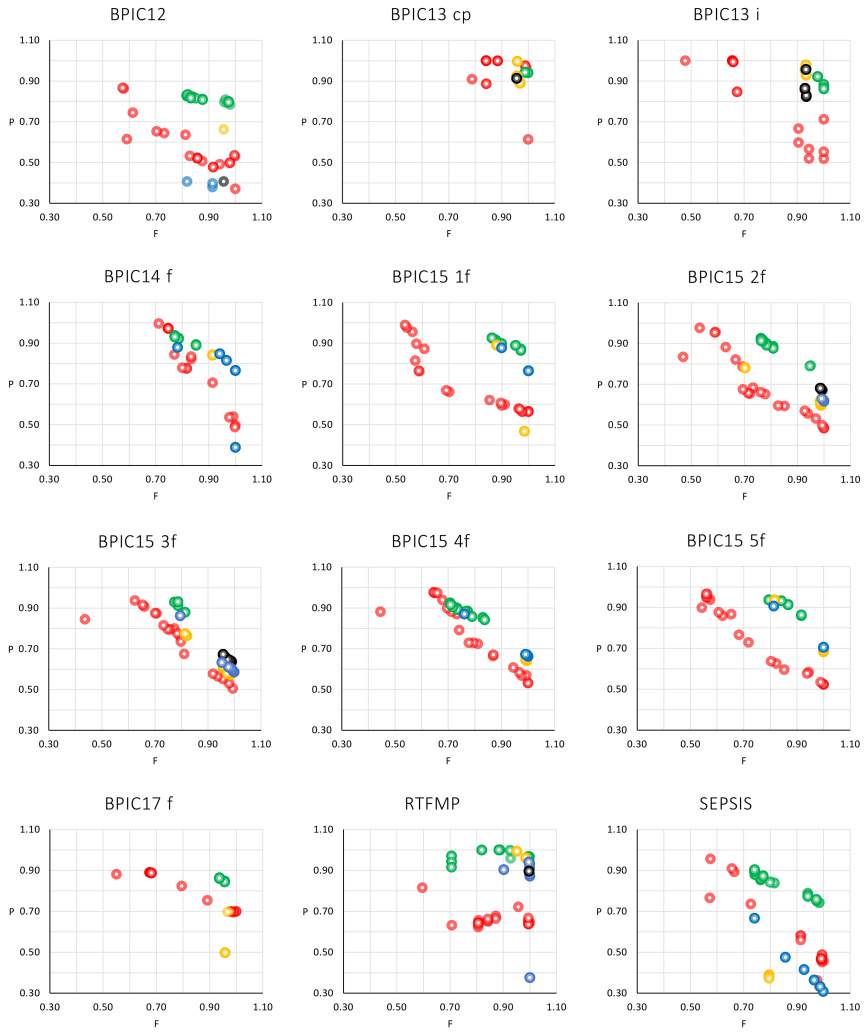


Fig. 8: Results of the hyperparameter-optimization for SM(green), IM(red), HM_6(black), S-HM_6(yellow), FO(blue). Horizontal and vertical axes report fitness and precision (respectively).

size), with less skipping edges and with clearly delimited loops, and is more accurate than the one produced by IM.

6 Conclusion and Future Work

The empirical findings show that Split Miner is a step forward towards more scalable and robust methods for automated discovery of business process models. Split Miner outperforms all baselines in terms of F-score and generalization on all twelve real-life event logs included in the evaluation. It produces models that are comparable in terms of size and control-flow complexity to those produced by the Inductive Miner and the Evolutionary Tree Miner, which produced the best results along these complexity

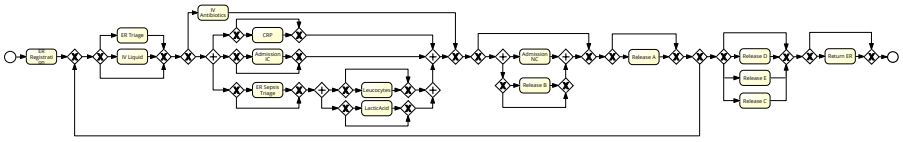


Fig. 9: Model discovered by IM from the Sepsis log.

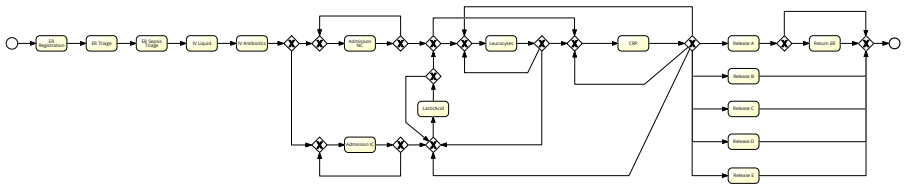


Fig. 10: Model discovered by SM from the Sepsis log.

measures in the majority of event logs. Furthermore, the execution times of the Split Miner were found to be at least three times faster (and up to 16 times faster) than the closest baseline across all event logs.

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 event logs. This same filtering step had to be applied for all baselines, since otherwise all baselines, as well as Split Miner, lead to low F-scores. 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.

Another direction for future work is to discover process models from more complex event logs than those that are typically extracted from enterprise information systems. For example, in the emerging field of Robotic Process Automation (RPA) [15], an open question is how to automatically discover repetitive routines – amenable for automation using RPA technology – from fine-grained event logs, such as clickstream data recording the interactions between process workers and enterprise applications. A major challenge is that in such fine-grained logs, the start and the end of repetitive routines is not clearly delimited and a process worker might sometimes be multitasking, which entails that events corresponding to multiple tasks might be interspersed with each other. To tackle this challenge, automated process discovery methods need to evolve from discovering models from collections of traces corresponding to well-delimited instances of a process, to discovering models from traces that contain tasks from multiple process instances (or even from multiple processes) mixed together.

Acknowledgments. This research is partly funded by the Australian Research Council (grant DP180102839) and the Estonian Research Council (grant IUT20-55).

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

References

1. A. Adriansyah, J. Muñoz-Gama, J. Carmona, B.F. van Dongen, and W.M.P. van der Aalst. Measuring precision of modeled behavior. *ISeB*, 13(1):37–67, 2015.

2. A. Adriansyah, B.F. van Dongen, and W.M.P. van der Aalst. Conformance checking using cost-based fitness analysis. In *Proc. of EDOC*. IEEE, 2011.
3. A. Augusto, R. Conforti, M. Dumas, M. La Rosa, and G. Bruno. Automated discovery of structured process models: Discover structured vs. discover and structure. In *Proc. of ER*, LNCS 9974. Springer, 2016.
4. Adriano Augusto, Raffaele Conforti, Marlon Dumas, and Marcello La Rosa. Split miner: Discovering accurate and simple business process models from event logs. In *Proceedings of the 17th IEEE International Conference on Data Mining*. IEEE Computer Society, 2017.
5. Adriano Augusto, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, Fabrizio Maria Maggi, Andrea Marrella, Massimo Mecella, and Allar Soo. Automated discovery of process models from event logs: Review and benchmark. *CoRR*, abs/1705.02288, 2017.
6. 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*, LNCS 7565. Springer, 2012.
7. J. S. Cardoso. Business process control-flow complexity: Metric, evaluation, and validation. *Int. J. Web Service Res.*, 5(2):49–76, 2008.
8. Weiru Chen, Jing Lu, and Malcolm Keech. Discovering exclusive patterns in frequent sequences. *International Journal of Data Mining, Modelling and Management*, 2(3):252–267, 2010.
9. R. Conforti, M. Dumas, L. García-Bañuelos, and M. La Rosa. BPMN Miner: Automated discovery of BPMN process models with hierarchical structure. *Inf. Syst.*, 56, 2016.
10. R. Conforti, M. La Rosa, and A.H.M. ter Hofstede. Filtering out infrequent behavior from business process event logs. *IEEE Trans. Knowl. Data Eng.*, 29(2), 2017.
11. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959.
12. Bolin Ding, David Lo, Jiawei Han, and Siau-Cheng Khoo. Efficient mining of closed repetitive gapped subsequences from a sequence database. In *Proc. of the International Conference on Data Engineering (ICDE)*, pages 1024–1035. IEEE, 2009.
13. M. Dumas, L. García-Bañuelos, and A. Polyvyanyy. Unraveling unstructured process models. In *BPMN Workshop*, volume 67 of *Lecture Notes in Business Information Processing*, pages 1–7. Springer, 2010.
14. M. Dumas, M. La Rosa, J. Mendling, R. Mäesalu, H.A. Reijers, and N. Semenenko. Understanding business process models: the costs and benefits of structuredness. In *Proc. of CAiSE*. Springer, 2012.
15. P. Harmon. Robotic process automation comes of age. *BPTrends Newsletter*, June 2017.
16. 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*, LNCS. Springer, 2013.
17. Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering block-structured process models from event logs containing infrequent behaviour. In Niels Lohmann, Minseok Song, and Petia Wohed, editors, *Business Process Management Workshops: BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers*, pages 66–78. Springer International Publishing, Cham, 2014.
18. Jing Lu, Weiru Chen, Osei Adjei, and Malcolm Keech. Sequential patterns postprocessing for structural relation patterns mining. *Strategic Advancements in Utilizing Data Mining and Warehousing Tech: New Concepts and Developments*, page 216, 2008.
19. Jing Lu, Weiru Chen, and Malcolm Keech. Graph-based modelling of concurrent sequential patterns. *Expl Adv in Interdiscip Data Mining and Anal: New Trends*, page 110, 2011.
20. Heikki Mannila, Hannu Toivonen, and A Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowl Discovery*, 1(3):259–289, 1997.
21. J. Mendling. *Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness*. Springer, 2008.
22. J. Mendling, H.A Reijers, and W.M.P van der Aalst. Seven process modeling guidelines (7PMG). *Information and Software Technology*, 52(2):127–136, 2010.
23. 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.
24. Jian Pei, Haixun Wang, Jian Liu, and et al. Discovering frequent closed partial orders from strings. *IEEE Transactions on Knowledge and Data Engineering*, 18(11):1467–1481, 2006.
25. A. Polyvyanyy. *Structuring process models*. Phd thesis, Universität Potsdam, 2012.
26. A. Polyvyanyy, L. García-Bañuelos, and M. Dumas. Structuring acyclic process models. *Inf. Syst.*, 37(6):518–538, 2012.
27. A. Polyvyanyy, L. García-Bañuelos, D. Fahland, and M. Weske. Maximal structuring of acyclic process models. *Comput. J.*, 57(1):12–35, 2014.
28. A. Polyvyanyy, J. Vanhatalo, and H. Völzer. Simplified computation and generalization of the refined process structure tree. In *Web Services and Formal Methods*, volume 6551 of *LNCS*, pages 25–41. Springer, 2010.
29. Y. Tong, L. Zhao, D. Yu, S. Ma, Z. Cheng, and K. Xu. Mining compressed repetitive gapped sequential patterns efficiently. *Advanced Data Mining and Applications*, pages 652–660, 2009.

30. W.M.P. van der Aalst. *Process Mining - Data Science in Action*. Springer, 2016.

31. W.M.P. 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.*, 23(3), 2011.

32. W.M.P. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. Knowl. Data Eng.*, 16(9), 2004.

33. Seppe KLM vanden Broucke and Jochen De Weerdt. Fodina: a robust and flexible heuristic process discovery technique. *Decision Support Systems*, 2017.

34. Hagen Völzer. A new semantics for the inclusive converging gateway in safe processes. In *International Conference on Business Process Management*, pages 294–309. Springer, 2010.

35. J. De Weerdt, M. De 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.*, 37(7), 2012.

36. A. Weijters and J. Ribeiro. Flexible Heuristics Miner (FHM). In *Proc. of CIDM*. IEEE, 2011.

37. Mohammed J Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1):31–60, 2001.

Adriano Augusto is joint Ph.D. student at the University of Tartu (Estonia) and The University of Melbourne (Australia). He graduated in Computer Engineering at Polytechnic of Turin (Italy) in 2016, presenting a master thesis in the field of process mining. His current interests are in automated process discovery and conformance checking.

Raffaele Conforti is a Lecturer in Information Systems with The University of Melbourne, Australia. Prior to that, he was a Post-Doctoral Research Fellow at the Queensland University of Technology. He conducts research on process mining and automation, with a focus on automated process discovery, quality improvement of process event logs and process-risk management.

Marlon Dumas is a Professor of Information Systems at University of Tartu, Estonia. Prior to this appointment he was faculty member at Queensland University of Technology and visiting researcher at SAP Research, Australia. His research interests span across the fields of software engineering, information systems and business process management. He is co-author of the textbook "Fundamentals of Business Process Management".

Marcello La Rosa is a Professor of Information Systems at The University of Melbourne, Australia. Prior to that, he held appointments at the Queensland University of Technology and at NICTA (now Data61). His research interests include process mining, consolidation and automation. He leads the Apromore Initiative (www.apromore.org), a strategic collaboration between various universities for the development of a process analytics platform, and co-authored the textbook "Fundamentals of Business Process Management".

Artem Polyvyanyy is a Senior Lecturer at the School of Computing and Information Systems, Melbourne School of Engineering, at the University of Melbourne, Australia. He received a PhD degree (Dr. rer. nat.) in the scientific discipline of Practical Computer Science from the University of Potsdam, Germany. His research interests include Distributed and Parallel Systems, Formal Methods, Information Systems, Software Engineering, Workflow Management, and Business Process Management. More recently, he has conducted research on process analysis, behavior abstraction in concurrent systems, process mining, and process querying.