An Approximate Inductive Miner

Jan Niklas van Detten
Celonis Labs GmbH & RWTH Aachen
Aachen, Germany
niklas.van.detten@rwth-aachen.de

Pol Schumacher
Celonis Labs GmbH
München, Germany
p.schumacher@celonis.com

Sander J. J. Leemans
RWTH Aachen
Aachen, Germany
s.leemans@bpm.rwth-aachen.de

Abstract—Process discovery algorithms extract process models from business process event logs. Existing discovery algorithms require upfront filtering, or specific parameter input, to produce models with balanced quality dimensions on real-life event logs. We propose the Approximate Inductive Miner (AIM) to fill this gap and offer an automated way to discover sound models in polynomial time complexity, without any pre-processing or mandatory parameter input. AIM uses the existing Inductive Miner framework and applies clustering techniques to recursively identify structures in the event log. It additionally performs an approximate parameter optimisation to dynamically suggest a suitable parameter. We compare AIM with existing discovery algorithms on synthetic and real-life event logs, and evaluate the quality of the integrated parameter suggestion. We find that AIM on its own produces sound models with low control flow complexity and high precision, even on complex event logs. Additionally, AIM is able to handle a vast range of event log properties, such as infrequent and incomplete behaviour, without requiring any human parameter input or upfront filtering.

Index Terms—process mining, process discovery, automated discovery, parameter approximation

I. Introduction

Modern business environments produce track records of process activities in event logs, which enable the empirical analysis of the underlying business process. An important step in process analysis is the discovery of a process model. Many algorithms exist for this purpose, each producing models with different properties on relevant quality dimensions.

Often used process model quality criteria include fitness, precision, simplicity, soundness and rediscoverability. Fitness describes a model's ability to represent the behaviour contained in the event log. Precision measures how well a model excludes behaviour dissimilar to the event log. Simplicity values a minimal control flow complexity and model size. The soundness property guarantees a model to be deadlock free and to not include any unreachable parts [1]. Rediscoverability is the ability to discover a process model from an event log that is equivalent to the original one [2].

In addition to these model-specific and log-based properties, there are further requirements on the discovery algorithm itself. The time and space complexity should scale well with increasing event log size and complexity. Additionally, it should be adaptable to different levels of noise, which can be found in real-life event logs [3]. An ideal process discovery algorithm should reliably produce good models on relevant quality metrics and possess these properties.

Some existing discovery algorithms adhere to the Inductive Miner framework (IM) introduced in [4]. IM splits an event log by recursively searching an activity partition and an operator, which describes the control flow relation between the resulting partition's parts. The combination of such an activity partition and operator is called a cut. The IM recursion continues by splitting the event log based on the detected cut and stops when a base case is reached, meaning that the event log cannot be split any further. A set of fall-through methods handles event logs that do not perfectly fit any of the available operators and are not a base case. These methods increasingly sacrifice precision to represent the log with the available operators. All implementations of IM guarantee model soundness.

Many algorithms, such as [5], [6] and [7], handle noise with filtering methods based on parameters. The flexibility introduced by these parameters is offset by the large range of input values that a user must consider. This problem can be addressed by performing a hyper-parameter optimisation at design time. The resulting default parameters can however be sub-optimal if an event log differs substantially from the ones used for the optimisation. A user might still need to try a set of models with different parameter choices and evaluate them, which can be time intensive. To the best of our knowledge, an automated and runtime-efficient parameter suggestion, directly integrated into the discovery process itself, is not offered yet by any process discovery algorithms.

In this paper we introduce the Approximate Inductive Miner (AIM), which provides such an integrated parameter suggestion. It investigates a state space for potential cuts in the event log under consideration of the available filter parameter space. These cuts are evaluated with heuristic quality measures, which estimate the adherence of the event log to each cut and the potential information loss induced by our filtering method. Clustering techniques are subsequently used to prune the state space and to select a cut with a filter parameter. These techniques are integrated into the IM recursion, as a replacement for all fall-through methods, to benefit from the framework's already proven guarantees.

We compare the process models produced by AIM to

those from existing discovery algorithms on 18 real-life event logs and use 480 synthetic event logs to observe the structural influence of the utilised clustering techniques. To evaluate the integrated parameter suggestions, we additionally perform a hyper-parameter optimisation for comparison.

The remainder of this paper is structured as follows. Section II contains relevant process mining concepts and notations. AIM itself is explained in Section III. The evaluation of AIM is done in Section IV, while Section V provides an overview about existing discovery algorithms. Section VI contains our final thoughts and a summary of our findings.

II. Preliminaries

An event log $L = [\sigma_1^{c_1}, \ldots, \sigma_n^{c_n}]$ is a multi-set of traces σ_i with multiplicities c_i. A trace σ consists of ordered activities. For a trace that contains exactly the activities a and b, in this order, we write $\langle a, b\rangle$, while an empty trace is represented by ϵ. The set of unique activities in the event log is called the alphabet Σ. The start activities $\text{start}(L) = \{a \in \Sigma | \exists \sigma \in L : \sigma = \langle a, \ldots\rangle\}$ and end activities $\text{end}(L) = \{a \in \Sigma | \exists \sigma \in L : \sigma = \langle \ldots, a\rangle\}$ are the sets of unique activities that appear in the first and last position of a trace in the event log. We write $|a|$ for the total occurrence count of an activity a across all traces of the event log. $|L|$ is the number of traces in L, while $||L||$ describes the sum of all activity occurrence counts.

A follows-graph of the event log contains a node for each activity in the alphabet. The directly follows-graph (DFG) of an event log has an edge $a \nrightarrow b$ if $\exists \sigma \in L : \sigma = \langle \ldots, a, b, \ldots\rangle$. The strictly indirectly follows-graph (IFG) contains an edge $a \nrightarrow^* b$ if there is a trace $\sigma \in L$ in which b does not directly follow a and $\sigma = \langle \ldots, a, \ldots, b, \ldots\rangle$. The eventually follows- graph (EFG) has an edge $a \nrightarrow^+ b$ if $a \nrightarrow b \vee a \nrightarrow^* b$. The edge frequencies $|a \nrightarrow b|$, $|a \nrightarrow^+ b|$ and $|a \nrightarrow^* b|$ indicate how often b directly, eventually or indirectly follows a.

A. Process Trees

Process models can be represented by process trees for alphabets Σ with $\tau \notin \Sigma$. They consist of leaf nodes with labels in $\Sigma \cup \{\tau\}$ and inner nodes with labels in $\{\rightarrow, \times, ||, \circlearrowleft\}$. Inner nodes are nodes with children in the tree structure, while leaf nodes are those without. Without loss of generality we only consider binary process trees, which means that each inner node has exactly two children [2]. Each process tree t can be translated into $\mathcal{L}(t)$, the language of traces it allows.

A leaf with an activity label $a \in \Sigma$ represents the execution of this activity, while a τ represents the empty trace. Therefore $\mathcal{L}(\tau) = \{\epsilon\}$ and $\mathcal{L}(a) = \{\langle a\rangle\}$. The label of an inner node in the process tree describes by which operator its children are related. We define the resulting languages with the notation from [8] and [9]. The exclusive choice operator $\times$ allows a trace of only one of its children to be used with $\mathcal{L}(\times(t_1, t_2)) = \mathcal{L}(t_1) \mid \mathcal{L}(t_2)$. The sequence operator $\rightarrow$ concatenates a trace from its first and second children with $\mathcal{L}(\rightarrow\ (t_1, t_2)) = \mathcal{L}(t_1) \cdot \mathcal{L}(t_2)$. The parallel operator $||$ allows any pair of traces from its children to be interleaved, as long as their partial order is kept, with $\mathcal{L}(||\ (t_1, t_2)) = \mathcal{L}(t_1) \sqcup\!\sqcup \mathcal{L}(t_2)$. The loop operator $\circlearrowleft$ allows the repetition of traces from its first children after using a trace from the second one with $\mathcal{L}(\circlearrowleft\ (t_1, t_2)) = \mathcal{L}(t_1)(\mathcal{L}(t_2)\mathcal{L}(t_1))^*$. The first child of a loop is called the body part and the second child the redo part.

Exclusive choice and loop nodes with τ as one of their children nodes create special structures. Exclusive choice nodes with one τ and one non-τ as children are called τ skips, and are used to describe optional behaviour with $\times(\tau, t)$ or $\times(t, \tau)$ where $t \neq \tau$. A loop node with a non-τ body part and a τ redo part is called a τ loop, which allows arbitrary repetitions of the body part with $\circlearrowleft\ (t, \tau)$ where $t \neq \tau$.

B. Inductive Miner

The Inductive Miner framework (IM) for binary cuts is shown in Algorithm 1. It recursively constructs a process tree out of an event log. In each recursion step a cut is detected and the event log is split until a base case is reached. If no cut can be found and no base case applies, a fall-through method is used. In the following we recall an IM implementation with the operator set $\{\rightarrow, \times, ||, \circlearrowleft\}$ from [4], which we call IMb.

Algorithm 1 IM framework from [4]

```
function im(L)
    bc ← basecase(L)
    if bc ≠ ∅ then
        return bc
    end if
    ⊕, Σ1, Σ2 ←findcut(L)
    if ⊕ ≠ ∅ then
        L1, L2 ←splitlog(L, ⊕, Σ1, Σ2)
        return ⊕(im(L1),im(L2))
    else
        return fallthrough(L)
    end if
end function
```

If the event log is not a recursion base case, a cut needs to be detected with findcut_{IMb}. A cut consists of a partition of the alphabet into Σ_1, Σ_2 and an operator $\oplus \in \{\rightarrow, \times, ||, \circlearrowleft\}$. Each operator leads to a unique set of DFG properties if the event log adheres to it, which is called the operators footprint. The DFG of the event log is used to check if any alphabet partition fits an operator's footprint, which is not guaranteed.

When a cut can be found, the event log is split with splitlog_{IMb} into the sub-logs L_1 and L_2. For an operator $\oplus \in \{||, \times\ \rightarrow\}$ each of the activities in Σ is assigned

to sub-log L_1 or L_2 if it is in Σ_1 or Σ_2. For the operator $\circlearrowleft$ additional trace splitting is necessary. Each trace $\langle \dots a, b \dots \rangle$ with $a \in \Sigma_1 \wedge b \in \Sigma_2$ or $a \in \Sigma_2 \wedge b \in \Sigma_1$ is split between a and b. Traces are then divided over L_1 and L_2, depending on which part of the partition includes each of their activities.

The recursion continues on the resulting sub-logs until a base case is found with basecase_{IMb}. A base case is reached when the event log is empty, or when its alphabet only contains a single activity. In this case it is not possible to split the event log, or its alphabet, any further. An empty event log results in a τ, while an event log that only contains the trace $\langle a \rangle$ produces an activity leaf a. If the event log contains a trace with at least one repetition of the remaining activity a, such as $\langle a, a \rangle$, a self loop $\circlearrowleft(a, \tau)$ is returned as a base case.

In case no cut can be found, for an event log that is not a base case, a set of fall-through methods is applied with fallthrough_{IMb}. These fall-through methods are used to fit the event log into the representational bias of the available operators. They also account for τ skips and τ loops, but become increasingly less precise. The last fall-through method is called a flower model, which allows for the arbitrary execution and repetition of the remaining activities.

C. Relation & Quality Estimates

Several alternative IM implementations have been defined [5], [10] that, if no cut can be found, attempt to find a most likely cut, based on relation estimates. The likelihood of a cut is established by these techniques by taking the average estimated likelihood of pairs of activities that get separated by the cut's partition. That is, for the operators $\{\rightarrow, \times, ||\}$, the set M of these pairs is

$$M(\oplus, \Sigma_1, \Sigma_2) = \{(\oplus, a, b) \mid a \in \Sigma_1 \wedge b \in \Sigma_2\}$$

For the loop operator, one needs to distinguish two groups of separated activity pairs: those that have a direct connection, and those that do not [10]. The direct connection can be estimated through the first and last activities of $\Sigma_2(S_2, E_2)$ and the start and end activities contained in $\Sigma_1(S_1, E_1)$:

$$M(\circlearrowleft, \Sigma_1, \Sigma_2) = \{(\circlearrowleft_s, a, b) \mid a \in E_1 \wedge b \in S_2\}$$
$$\cup \{(\circlearrowleft_s, a, b) \mid a \in E_2 \wedge b \in S_1\}$$
$$\cup \{(\circlearrowleft_i, a, b) \mid a \in \Sigma_1 \setminus (S_1 \cup E_2) \wedge b \in \Sigma_2 \setminus (S_2 \cup E_2)\}$$

III. Approximate Inductive Miner

In this section we introduce the Approximate Inductive Miner (AIM). AIM uses a recursive strategy to discover a process tree based on a given event log and filter parameter space. In each recursion step, the best cut and filter parameter is sought. To select such a cut, AIM first determines the state space of potential cuts across the available parameter space and calculates a quality estimate for each of them. Second, AIM adjusts these cut quality estimates to account for the information loss induced during filtering. Third, AIM prunes this state space efficiently with the application of clustering techniques to select a cut. Finally, the full AIM algorithm is specified and its guarantees and properties are discussed.

A. Cut State Space

First, we describe the state space S_L of cuts that we consider in AIM, for a given event log L, filter parameter space $F \subseteq [0, 1]$, and a cut quality estimate function $Q(L, \oplus, a, b)$. The cuts come in two types: cuts with non-empty partition parts, and τ structure cuts, with one empty partition part. We assume an event log filtering function $\text{filter}(L, f)$ to be available with $f \in [0, 1]$. Then, the best cut $(\oplus, \Sigma_1, \Sigma_2)$ from S_L is

$$\underset{(\oplus, \Sigma_1, \Sigma_2) \in S_L}{\arg\max} \max_{f \in F} Q(\text{filter}(L, f), \oplus, \Sigma_1, \Sigma_2) \tag{1}$$

We now define the cuts in S_L for each operator and their corresponding cut quality estimates. A sequence cut $(\rightarrow, \Sigma_1, \Sigma_2)$ denotes that the activities in Σ_1 are executed exclusively before the activities in Σ_2. Let Σ_f be the alphabet of $\text{filter}(L, f)$. Then, the potential sequence cuts in S_L are

$$\{(\rightarrow, \Sigma_1, \Sigma_f \setminus \Sigma_1) \mid f \in F \wedge \Sigma_1 \subsetneq \Sigma_f \wedge \Sigma_1 \neq \emptyset\}$$

The cut quality estimate $Q(L, \rightarrow, \Sigma_1, \Sigma_2)$ is defined as the mean of the relation estimate multi-set with

$$Q(L, \rightarrow, \Sigma_1, \Sigma_2) = \mu_{(\rightarrow, a, b) \in M(\rightarrow, \Sigma_1, \Sigma_2)} R(L, \rightarrow, a, b)$$
$$R(L, \rightarrow, a, b) = \frac{|a \twoheadrightarrow^+ b|}{|a \twoheadrightarrow^+ b| + |b \twoheadrightarrow^+ a| + 1}$$

An exclusive choice cut $(\times, \Sigma_1, \Sigma_2)$ denotes that in each trace only activities from one partition part are executed. Let a be an arbitrary activity appearing in the most traces of L with $a = \arg\max_{a \in \Sigma} |[\sigma \in L \mid \sigma = \langle \dots, a, \dots \rangle]|$. Then, the potential, non-symmetrical exclusive choice cuts in S_L are

$$\{(\times, \Sigma_1, \Sigma_f \setminus \Sigma_1) \mid f \in F \wedge \Sigma_1 \subset \Sigma_f \wedge a \notin \Sigma_1 \wedge \Sigma_1 \neq \emptyset\}$$

Notice that a prevents symmetrical cuts. The cut quality estimate $Q(L, \times, \Sigma_1, \Sigma_2)$ is defined analogue to $Q(L, \rightarrow, \Sigma_1, \Sigma_2)$, based on $R(L, \times, a, b)$ with

$$Q(L, \times, \Sigma_1, \Sigma_2) = \mu_{(\times, a, b) \in M(\times, \Sigma_1, \Sigma_2)} R(L, \times, a, b)$$
$$R(L, \times, a, b) = \frac{1}{|a \twoheadrightarrow^+ b| + |b \twoheadrightarrow^+ a| + 1}$$

A concurrent cut $(||, \Sigma_1, \Sigma_2)$ describes the interleaved concurrent execution of the activities in the different parts of the alphabet partition. The potential, non-symmetrical concurrent cuts in S_L, analogue to the exclusive choice cuts, are

$$\{(||, \Sigma_1, \Sigma_f \setminus \Sigma_1) \mid f \in F \wedge \Sigma_1 \subset \Sigma_f \wedge a \notin \Sigma_1 \wedge \Sigma_1 \neq \emptyset\}$$

The cut quality estimate for concurrent cuts $Q(L, ||, \Sigma_1, \Sigma_2)$ applies a bias l which measures the average excess trace length in comparison to the alphabet size with

$$Q(L, ||, \Sigma_1, \Sigma_2) = \mu_{(||,a,b)\in M(||,\Sigma_1,\Sigma_2)} R(L, ||, a, b) \cdot (1 - l)$$

$$R(L, ||, a, b) = \frac{2 \cdot |a \nrightarrow b| \cdot |b \nrightarrow a|}{|a \nrightarrow b|^2 + |b \nrightarrow a|^2 + 1}$$

$$l = \min(1, \max(\mu([|\sigma| \mid \sigma \in L]) - |\Sigma|, 0) \cdot |\Sigma|^{-1})$$

A loop cut $(\circlearrowleft, \Sigma_1, \Sigma_2)$ denotes the optional repetition of the activities in Σ_1, as long as the activities in Σ_2 are executed before. The set of potential loop cuts in S_L is

$$\{(\circlearrowleft, \Sigma_1, \Sigma_f \setminus \Sigma_1) \mid f \in F \wedge \Sigma_1 \subset \Sigma_f \wedge a \notin \Sigma_1 \wedge \Sigma_1 \neq \emptyset\}$$

The quality estimate for loop cuts $Q(L, \circlearrowleft, \Sigma_1, \Sigma_2)$ requires the assumed start and end activities of the redo part of the loop. For a given loop cut $(\circlearrowleft, \Sigma_1, \Sigma_2)$ we assume an activity $a \in \Sigma_2$ to be part of the start activities of $\Sigma_2(S_2)$ if $\text{argmax}_{b\in\Sigma_f}(R(L, \circlearrowleft_s, b, a)) \in \text{end}(L) \cap \Sigma_1$. Similarly, a is part of the end activities E_2 if $\text{argmax}_{b\in\Sigma}(R(L, \circlearrowleft_s, a, b)) \in \text{start}(L) \cap \Sigma_1$. The cut quality estimate is then

$$Q(L, \circlearrowleft, \Sigma_1, \Sigma_2) = \mu_{(\oplus,a,b)\in M(\circlearrowleft,\Sigma_1,\Sigma_2)} R(L, \oplus, a, b) \cdot l$$

$$R(L, \circlearrowleft_s, a, b) = \frac{2 \cdot |a \nrightarrow b| \cdot |b \nrightarrow^+ a|}{|a \nrightarrow b|^2 + |b \nrightarrow^+ a|^2 + 1}$$

$$R(L, \circlearrowleft_i, a, b) = \frac{2 \cdot |a \nrightarrow^* b| \cdot |b \nrightarrow^* a|}{|a \nrightarrow^* b|^2 + |b \nrightarrow^* a|^2 + 1}$$

A τ skip cut $(\times, \Sigma, \emptyset)$ denotes the optional execution of the activities in Σ. We define the quality estimate for the τ skip cuts $\{(\times, \Sigma_f, \emptyset) \mid f \in F\}$ in S_L as

$$Q(L, \times, \Sigma, \emptyset) = |\{\sigma \in L \mid \sigma = \epsilon\}| \cdot |L|^{-1}$$

A τ-loop cut $(\circlearrowleft, \Sigma_f, \emptyset)$ denotes the optional repetition of the activities in Σ. For the τ-loop cuts $\{(\circlearrowleft, \Sigma_f, \emptyset) \mid f \in F\}$ in S_L we define the quality estimate

$$Q(L, \circlearrowleft, \Sigma, \emptyset) = \mu(M(R, L, \circlearrowleft, \Sigma, \Sigma)) \cdot l$$

B. Filter Parameter Adjustment

Picking the cut with the highest quality estimate in our state space can lead to excessive filtering. Removing activities from the event log not only increases the chance to find a near-perfect cut, but also induces information loss. To prevent excessive filtering we therefore adjust the quality estimate for each cut in S_L with the factor N_f. The adjusted quality estimate not only takes into account how good a cut fits the potentially filtered event log, but also expresses how much relative information loss was induced by the filtering. The best cut in S_L, in extension to equation (1), is therefore

$$\underset{(\oplus,\Sigma_1,\Sigma_2)\in S_L}{\arg\max} \max_{f\in F} Q(\text{filter}(L, f), \oplus, \Sigma_1, \Sigma_2) \cdot N_f \quad (2)$$

The filtering method used in our implementation removes activities that appear in relatively few traces, compared to the activity contained in the maximum number of traces. Let count(a) be the number of traces that contain the activity a with $\text{count(a)} = |[\sigma \in L \mid \sigma = \langle \ldots, a, \ldots \rangle]|$. An activity is subsequently removed from the event log by $\text{filter}_{AIM}(L, f)$ if $\text{count}(a) < \arg\max_{b\in\Sigma} \text{count}(b) \cdot f$. To capture the information loss induced by $\text{filter}_{AIM}(L, f)$, we measure the relation between the average trace length in the event log, before and after filtering. N_f is therefore $||L_f|| \cdot ||L||^{-1}$. Note that the choice of the available filter parameter space can be used to include knowledge about the event log into the discovery process, or to enforce requirements regarding the minimal or maximal amount of filtering.

C. State Space Pruning

The cut with the highest adjusted quality estimate in S_L provides a conceptual trade off. This cut can however not be determined effectively. Our cut state space grows exponentially in the size of Σ, irrespective of the utilized parameter space F. The exhaustive evaluation of all cuts in S_L is therefore not feasible and significant pruning of S_L is required instead.

The quality estimate for a cut $(\oplus, \Sigma_1, \Sigma_2)$ with non-empty partition parts depends on the average value of $R(L, \oplus, a, b)$, for the activity pairs (a, b) in $M(\oplus, \Sigma_1, \Sigma_2)$. These activity pairs always contain one activity from each part of the alphabet partition. Cuts with high quality estimates in S_L therefore correspond to high values of $R(L, \oplus, a, b)$ for $a \in \Sigma_1 \wedge b \in \Sigma_2$ or $b \in \Sigma_1 \wedge a \in \Sigma_2$. We subsequently approximate such cuts, by using $R(L, \oplus, a, b)$ as a distance measure for clustering. We apply k-means, with a fixed cluster count of two, for each $\oplus \in \{\rightarrow, \times, ||\}$ and $f \in F$. The clusters detected on $R(L_f, \oplus, a, b)$ correspond to the alphabet partition $\Sigma_{\oplus,f,1}, \Sigma_{\oplus,f,2}$. A high relation estimate between two activities increases the probability of them ending up in different parts of $\Sigma_{\oplus,f,1}, \Sigma_{\oplus,f,2}$. Note that $R(L_f, \oplus, a, b)$ potentially contains arbitrary many clusters for $\oplus \in \{\times, ||, \rightarrow\}$. Clustering fails if there are less then two, which can only happen if all entries in $R(L_f, \oplus, a, b)$ are identical. In this case, all activities have the same relation estimate to each other and an arbitrary alphabet partition can be used. We subsequently reduce our search space S_L by excluding all cuts with the symmetric operators $\oplus \in \{|| \ \times\}$ and non-empty partition parts, except for the ones in $\{(\oplus, \Sigma_{\oplus,f,1}, \Sigma_{\oplus,f,2}) \mid f \in F \wedge \oplus \in \{||, \times\}\}$. For the non-symmetric sequence operator we need to consider the ordering of the partition and therefore exclude all sequence cuts, except $\{(\rightarrow, \Sigma_{\rightarrow,f,1}, \Sigma_{\rightarrow,f,2}), (\rightarrow, \Sigma_{\rightarrow,f,2}, \Sigma_{\rightarrow,f,1}) \mid f \in F\}$.

The quality estimate for a loop cut $(\circlearrowleft, \Sigma_1, \Sigma_2)$ depends on the assumed start and end activities for each part of the alphabet partition (S_1, E_1, S_2, E_2), and the relation estimates $R(L, \circlearrowleft_i, a, b)$ and $R(L, \circlearrowleft_s, a, b)$. The clustering based pruning is therefore not applicable here and a different strategy is needed to detect high quality loop cuts. We attempt to find such a cut by first determining the start and end activities of each partition part, effectively

establishing the borders between them. Afterwards, the remaining activities are assigned to Σ_1 and Σ_2, based on their relation to the already determined start and end activities. This process is done for all $f \in F$.

We start by sorting the activities in $\text{start}(L_f) \cup \text{end}(L_f)$ by their occurrence count in descending order. We subsequently assign each activity a in this ordered set to $\Sigma_{\circlearrowleft,f,1}$. If $a \in \text{start}(L_f)$ holds, we additionally assign a to S_1 and the activity $b = \text{argmax}_{b \in \Sigma_f}(R(L_f, \circlearrowleft_s, b, a))$ to $\Sigma_{\circlearrowleft,f,2}$ and E_2. Similarly, if $a \in \text{end}(L_f)$ holds, we assign a to E_1 and the activity $b = \text{argmax}_{b \in \Sigma_f}(R(L_f, \circlearrowleft_s, a, b))$ to $\Sigma_{\circlearrowleft,f,2}$ and S_2. All activities that remain unassigned after this first assignment process are assigned to the same part of the loop, as the activity that they are the most connected with, according to $R_(L_F, \circlearrowleft_s, a, b)$. We subsequently reduce S_L by removing all loop cuts with non-empty partition parts except those in $\{(\circlearrowleft, \Sigma_{\circlearrowleft,f,1}, \Sigma_{\circlearrowleft,f,2}) \mid f \in F\}$.

Our pruned search space is significantly smaller than the original one and contains exactly seven cuts for each parameter value $f \in F$. There is one cut with non-empty partition parts for each of the available operators, with an additional cut for the non-symmetric sequence operator. Additionally, the τ skip cut and the τ loop cut remain. This pruned state space only grows linear in the size of the selected parameter space $|F|$ and not exponential in the size of the alphabet $|\Sigma|$.

D. Implementation

We integrate our approaches into IM with the following implementation. It uses BaseCase_{AIM}, FindCut_{IMb}, FallThrough_{AIM} and SplitLog_{IMb}, in accordance with Algorithm 1. FallThrough_{AIM} performs our state space pruning to find a cut and to suggest the corresponding filter parameter. As shown in Algorithm 2, this requires a new log splitting method SplitLog_{AIM}. We choose the parameter space of discrete 0.1 steps with $F = \{\frac{x}{10} \mid x \in \mathbb{N} \wedge x < 10\}$.

Algorithm 2 AIM Fallthrough

```
function fallthrough_AIM(L)
    ⊕, Σ1, Σ2, f ← findcut_AIM(L)
    L1, L2 ← splitlog_AIM(Lf, ⊕, Σ1, Σ2)
    return ⊕(im(L1), im(L2))
end function
```

Notice that the high-level method SplitLog_{IMb} used in the IM framework is not changed. The lower-level function SplitLog_{AIM} described here is only used in FallThrough_{AIM} from Algorithm 2. SplitLog_{AIM} extends SplitLog_{IMb} with cuts that do not fit the potentially filtered event log perfectly. It additionally covers τ loops and τ skips. We remove all empty traces before applying the splitting from IMb. This way we avoid an undesirable accumulation of empty traces across multiple recursion steps with such cuts. For τ loops we additionally split all traces $\langle \ldots, a, b, \ldots \rangle$ between a and b if $a \in \text{end}(L_f) \wedge b \in \text{start}(L_f)$.

BaseCase_{AIM} returns the base case τ if the event log is empty. If the event log contains more then one unique activity, no base case applies. If the event log contains only one unique activity a, BaseCase_{AIM} returns either $\times(a, \tau)$, a or $\circlearrowleft(a, \tau)$, depending on the tightest structure that fits the most traces. We define $Q_\times(L) = |[\sigma \in L \mid \sigma = \epsilon]|$, $Q_a(L) = |[\sigma \in L \mid \sigma = \langle a \rangle]|$ and $Q_\circlearrowleft(L) = |[\sigma \in L \mid \sigma \neq \langle a \rangle \wedge \sigma \neq \epsilon]|$. $\times(a, \tau)$, a and $\circlearrowleft(a, \tau)$ are chosen depending on the maximum value of $Q_\times(L), Q_\circlearrowleft(L)$ and $Q_a(L)$. In case of equality we use the default base case order a, $\times(a, \tau)$, $\circlearrowleft(a, \tau)$.

E. Complexity & Guarantees

AIM provides the same rediscoverability guarantees as IMf, which can be proven irrespective of the utilised fallthrough methods [2, Section 6.2.4]. All models produced by AIM are guaranteed to be sound due to the use of process trees [2]. Additionally, we sketch a proof for the worst case time complexity of AIM.

Each recursion step of AIM either detects a cut that reduces the alphabet size, or results in a τ-skip, or τ-loop. Due to SplitLog_{AIM} there can be no consecutive recursion steps, of τ skips or τ-loops. Therefore, the amount of recursion steps of AIM is in $O(|\Sigma|)$.

The execution of BaseCase_{AIM} only contains the calculation of $Q_\times$, Q_a and $Q_\circlearrowleft$, which can be done in $O(||L||)$. The log splitting SplitLog_{AIM} can also be done in $O(||L||)$, while the cut detection FindCut_{IMb} is of complexity $O(|\Sigma|^2)$ [2]. Our activity filtering is done in $O(||L||)$ and the relation estimates between all activities are determined in $O(|\Sigma|^2)$, for every potential filter parameter. Additionally, clustering is applied for each filter parameter with a time complexity of $O(g \cdot r \cdot |\Sigma|^2)$ where g is the number of iterations required for a single run of k-means and r the number of repetitions per recursion step [11]. A single recursion step is therefore of complexity $O(|F| \cdot (||L|| + g \cdot r \cdot |\Sigma|^2))$. It follows a total worst case time complexity of $O(|\Sigma| \cdot |F| \cdot (||L|| + g \cdot r \cdot |\Sigma|^2))$. Note that $|F|$, r and g are all chosen constant which have the values of ten, twelve and 300 in our implementation respectively.

IV. Evaluation

In this section we evaluate AIM threefold. First, we observe the influence of various event log properties, on the performance of our cut detection strategy, with different quality estimates. Then, we compare the integrated parameter suggestion of AIM with a hyper-parameter optimisation on the same set of parameters. Finally, we compare the performance of AIM with the performances of existing discovery algorithms.

A. Structure Detection

We use 480 synthesised event logs from the Process Discovery Contest 2022 (PDC) [12] to systematically

observe any structural weakness of AIM. That is, we evaluate the quality of the clustering based cut detection in isolation by setting $F = \{0.0\}$. The PDC event logs are all generated from a configurable model, which makes it possible to evaluate the influence of each model property individually. There are long term dependencies (DP), optional choice constructs (OR), routing constructs (RC), optional activities (OA) and duplicated activities (DA), which are either included (-Y) or not (-N). The model can have loops with single entry points (LC-S), multiple entry points (LC-M), or no loops at all (LP-N). Each possible combination of these configurations results in a ground truth model, which is used to generate a training event log, with a thousand random walks through the model. These event logs can either be noise free (NL-N), or have events removed (NL-R), moved (NL-M) or added (NL-A). A combination of events being removed, moved and added is possible as well (NL-Y). The fitness of a thousand trace pairs is measured on the original and the discovered model. The accuracy is defined as the percentage of trace pairs, for which the same trace fits both models better. Figure 1 shows the resulting accuracy distributions for each property configuration.

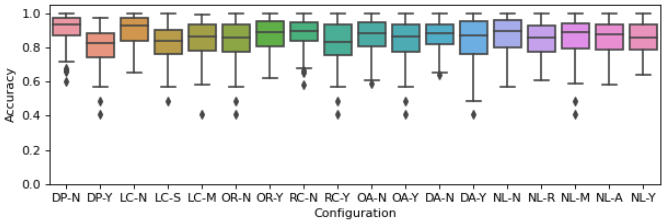

Fig. 1. AIM PDC 2022 classification accuracy with $F = \{0.0\}$

AIM achieves an average accuracy of 0.86. The biggest drop in performance is caused by including long term dependencies in the model (DP). The second highest drop is due to the general inclusion of loops (LC), with only a small difference between simple and complex loops. The remaining properties cause less significant differences in the accuracy distribution. We believe the performance loss for long term dependencies to be caused by the information loss induced during the event log splitting: any potential dependencies between events that end up in different sub-logs are not taken into account in subsequent recursion steps. The performance drop under the inclusion of loop constructs of any complexity is an indication that our loop detection strategy has some room for improvement left. Our results clearly showcase, that AIM successfully accounts for a vast range of model and event log properties, with the discussed limitations.

B. Parameter Choice

We use the 18 real-life event logs from the Business Process Intelligence Challenge and other public benchmarks [13]–[21], to quantitatively evaluate the integrated parameter suggestion. First, AIM is applied with $F = \{\frac{x}{10} \mid x \in \mathbb{N} \wedge x < 10\}$. We additionally use AIM with each of the filter parameter values $f \in F$ individually, which represents a hyper-parameter optimisation. We subsequently measure alignment based fitness [22] and precision [23], as well as the model size [24].

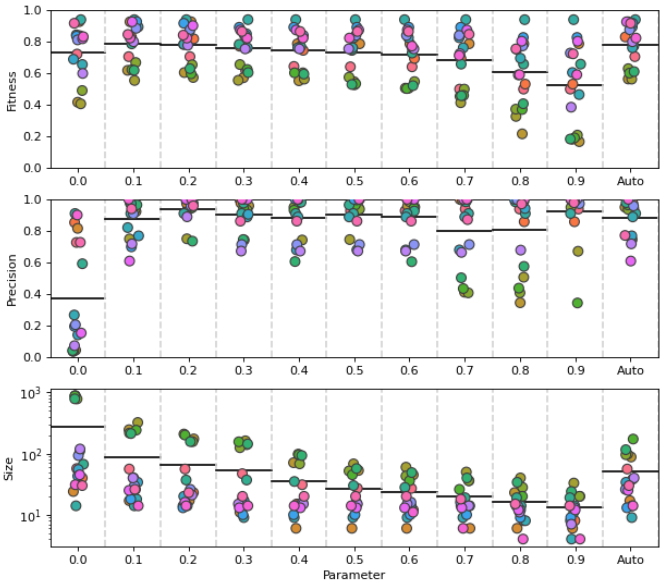

Fig. 2. Hyper-Parameter optimisation of AIM

AIM with the integrated parameter suggestion, on average, achieves a fitness of 0.78, a precision of 0.88 and a model size of 51 nodes across the 18 event logs. The results for all investigated parameter spaces and event logs can be seen in Figure 2. The average fitness of the parameter suggestion is optimal across all investigated parameter spaces, with the hyper-parameter optimisation yielding an average value of 0.77 for $F = \{0.2\}$. For the same parameter space $F = \{0.2\}$ the precision improves in comparison to the integrated parameter suggestion, with an average value of 0.93. This however comes at the cost of larger models with 70 nodes on average. The Hyper-parameter optimisations of the model size leads to the parameter space of $F = \{0.9\}$. On average, it produces smaller models with only 13 nodes and a high precision of 0.92, but causes a worse fitness of 0.53 in return. Overall, the results showcase that the integrated parameter suggestion is able to dynamically balance the quality metrics of fitness, precision and model size, without requiring a time intensive hyper-parameter optimisation. Furthermore, searching all parameter took five days, while AIMs integrated parameter suggestion only run took half an hour in contrast.

C. Overall Performance

We use the public benchmark event logs from [13]–[21] to compare AIM with the existing discovery algorithms Inductive Miner infrequent (IMf) [7], Inductive Miner incomplete (IMc) [10] and Split Miner (SM) [6]. Unfortunately, we could not include the Probabilistic Inductive Miner (PIM) [5] in this comparison, as its core concepts are not published and its implementation is not publicly available for research purposes. We apply IMf, IMc and SM with their default parameters and AIM with the integrated parameter suggestion. All experiments are executed on the same hardware, providing an Intel Core I5-8625-U processor with exclusive access to 16GB of working memory. We choose time limits of four hours for the discovery of a single model and twelve hours for the calculation of each metric. We again measure alignment based fitness [22] and precision [23] in addition to model size [24].

During the model discovery we observed eight discovery timeouts for IMc. We believe the reason for these timeouts to be the exponential time complexity of IMc. All other algorithms discovered models in the given time limit.

Three models of SM could not be evaluated, because they were not sound. The remaining algorithms produce only sound models, because they all represent models with process trees.

The model evaluation caused six timeouts for IMc. The high amount of τ structures in the discovered models is a potential reason for the excessive amount of time needed for the model evaluation. SM caused three timeouts during the model evaluation. We assume the reason for these evaluation timeouts to be the size of the discovered models. IMf caused a similar problem, with evaluation timeouts on the six largest models. AIM was the only algorithm to discover sound models for all logs, that could also be evaluated in the given time limit.

Table I shows all the results. We observe the number of pareto-optimal models for the measured quality metrics. AIM achieves the best results (17), followed by SM (12), then IMf (6) and lastly IMc (2). The majority of models with the highest precision per event log are found by AIM (11/18) and SM (6/18). IMf only produces the most precise model for the event log from [15], while IMc never finds the most precise model. The models with the highest fitness are found by SM (8/18), AIM (7/18) and IMc (3/18). The smallest models are almost exclusively found by AIM (16/18), with a clear difference to the other algorithms. This is especially apparent for the event logs from BPIC 2015 [16], which have more than 350 unique activities. Overall, AIM trades a little fitness for smaller and more precise models in comparison to SM, IMf and IMc. Furthermore, it provides the unique ability to handle event logs with a high number of unique activities and removes the need for a user to choose a parameter.

As limitations, we note that some of the event logs from

TABLE I
AIM, SM, IMf & IMc on Benchmark Event Logs

Event Log	Algorithm	Precision	Fitness	Size
BPIC 2012_1 [13]	AIM	0.9405	0.7044	56
	IMf	0.5126	0.8004	59
	SM	0.9604	0.8976	47
	IMc	nan	nan	nan
BPIC 2013_1 [14]	AIM	0.9758	0.8293	26
	IMf	0.9717	0.9122	32
	SM	0.7771	0.9669	39
	IMc	nan	nan	nan
BPIC 2013_2 [15]	AIM	0.9634	0.9254	17
	IMf	0.9764	0.7591	18
	SM	0.9033	0.9386	18
	IMc	0.9371	0.8206	35
BPIC 2015_1 [16]	AIM	0.9501	0.5627	88
	IMf	nan	nan	588
	SM	nan	nan	1078
	IMc	nan	nan	nan
BPIC 2015_2 [16]	AIM	0.7418	0.5620	98
	IMf	nan	nan	531
	SM	nan	nan	1058
	IMc	nan	nan	nan
BPIC 2015_3 [16]	AIM	0.9359	0.6282	96
	IMf	nan	nan	758
	SM	nan	nan	926
	IMc	nan	nan	nan
BPIC 2015_4 [16]	AIM	0.9643	0.5999	173
	IMf	nan	nan	526
	SM	nan	nan	1072
	IMc	nan	nan	nan
BPIC 2015_5 [16]	AIM	0.7490	0.6095	116
	IMf	nan	nan	614
	SM	nan	nan	855
	IMc	nan	nan	nan
BPIC 2017_1 [17]	AIM	0.9786	0.7724	38
	IMf	0.7623	0.8847	45
	SM	0.8742	0.9018	62
	IMc	nan	nan	nan
BPIC 2019_1 [18]	AIM	0.8881	0.7608	9
	IMf	nan	nan	94
	SM	nan	nan	103
	IMc	nan	nan	nan
BPIC 2020_1 [19]	AIM	0.7682	0.8930	34
	IMf	0.3584	0.9165	61
	SM	0.7953	0.9692	56
	IMc	nan	nan	nan
BPIC 2020_2 [19]	AIM	0.9999	0.9160	13
	IMf	0.2709	0.9529	33
	SM	0.9097	0.9960	29
	IMc	nan	nan	nan
BPIC 2020_3 [19]	AIM	0.9577	0.8753	26
	IMf	0.1340	0.8749	83
	SM	0.8676	0.9818	64
	IMc	nan	nan	nan
BPIC 2020_4 [19]	AIM	0.7173	0.7988	40
	IMf	0.1835	0.8323	108
	SM	0.7927	0.9664	111
	IMc	nan	nan	nan
BPIC 2020_5 [19]	AIM	0.6084	0.9252	25
	IMf	0.2605	0.9335	36
	SM	0.8863	0.9939	32
	IMc	0.1919	0.9995	60
SEPSIS [21]	AIM	1.0	0.8223	14
	IMf	0.6578	0.9726	26
	SM	0.9468	0.9940	22
	IMc	0.8042	0.9997	42
RTFMP [20]	AIM	0.7707	0.9160	30
	IMf	0.8024	0.9028	31
	SM	0.9955	0.7899	33
	IMc	nan	nan	nan

[13]–[21] were part of the hyper-parameter optimisation, used to select default parameters for SM in [6], which may have given SM a slight advantage. All IM implementations inherently compress event logs into the representational bias of the available operators. The evaluations of AIM, IMf and IMc therefore measure to some extent the adherence of the event logs to these operators, which might limit generalisability. The usage of k-means introduces non deterministic behaviour. We did however not observe any differences in the discovered models for three different random seeds in our experiments.

V. Related Work

Many process discovery techniques have been proposed in literature; for a survey on recent process discovery techniques, please refer to [24].

Techniques that do not guarantee sound models include the Structured Heuristic Miner 6.0, which discovers a model based on a heuristic approach without any restriction to the models soundness [25]. The algorithm adjusts the resulting model in an attempt to make it sound, which is not guaranteed to succeed. The method provides two noise parameters.

The Split Miner enriches the directly follows graph of an event log with gateways that describe the control flow relation between activities [6]. Two parameters are provided that allow the user to decide how much filtering should be employed and to specify how much parallelism is contained in the event log. Default parameters are provided by a hyper-parameter optimisation. The approach does not guarantee the resulting models to be sound, but at least deadlock free.

Some process discovery algorithms utilize concepts of integer linear programming [26] and region theory [27]. Others focus specifically on special structures, such as duplicated activities [28], routing constructs [29] and loops [30]. However, none of these techniques guarantee to return a sound model.

Techniques that guarantee sound models include the Evolutionary Tree Miner, which is a genetic algorithm that evolves a population of process models based on the quality metrics fitness, precision, generalization and simplicity [31]. It offers the possibility to specify a level of importance for each of these quality metrics. A termination condition allows to instruct the miner to keep searching for a model that satisfies this condition, however this process can take an prohibitively long time on real-life logs.

The Inductive Miner Infrequent (IMf) is an implementation of the Inductive Miner framework (IM) that handles noise with conditional filtering [7]. When no cut can be found, parameterized filtering is applied and the cut detection is repeated. Fall-through methods are only used if the cut detection after the filtering fails as well. AIM replaces all fall-through methods with the clustering based cut detection and automatically determines the filter parameter value.

The Inductive Miner Incomplete (IMc) also adheres to the IM framework and handles incomplete event logs [10] with the usage of probabilistic cut quality estimates. It searches for the cut with the highest quality estimate through optimization techniques, but nevertheless suffers from the exponential number of potential cuts. Fall-through methods are only used if no cut can be found above a given threshold, which needs to be specified as an input parameter. AIM reuses the set of relevant activity pairs $M(\oplus, \Sigma_1, \Sigma_2)$, as described in Section II-C. AIM however chooses different quality estimates to address a range of event log properties in polynomial runtime.

The Probabilistic Inductive Miner (PIM) adapts the cut detection from IMc by using heuristic cut quality estimates [5] to address the cut search, though this remains exponential in worst case. PIM also introduces a new activity filtering method that still requires a parameter input. Unfortunately, no implementation of the approach has been made public for research purposes. AIM uses different cut quality estimates, which include measures for τ-loops and τ-skips, and guarantees a polynomial worst case runtime. AIM additionally provides a different filtering method with parameter suggestions.

Existing applications of clustering techniques in process mining focus on detecting trace clusters to subsequently discover multiple models [32]. Activity clustering is mostly used to prepare a decomposed model discovery [33]. To the best of our knowledge, none of the state-of-the-art discovery algorithms utilizes clustering to select model structures.

VI. Conclusion

In this paper we introduced the Approximate Inductive Miner (AIM), which utilizes clustering techniques to investigate an exponentially growing state space in polynomial runtime. We additionally proposed an integrated parameter suggestion, based on novel quality estimates, under consideration of the information loss induced by filtering the event log. Our evaluation showed that AIM successfully balances fitness, precision and model size in a given parameter space, without requiring human guidance or a time intensive hyper-parameter optimisation. Our cut quality estimates consistently handled challenging event log properties, such as infrequent or incomplete behaviour. We compared AIM to existing discovery algorithms and observed its unique ability to handle more than 350 unique activities per event log, without suffering from any comparative performance drawbacks.

References

[1] T. Murata, "Petri nets: Properties, analysis and applications," Proc. IEEE, vol. 77, no. 4, pp. 541–580, 1989.

[2] S. J. J. Leemans, "Robust process mining with guarantees," in Business Process Management (BPM 2018), vol. 2196 of CEUR Workshop Proceedings, pp. 46–50, CEUR-WS.org, 2018.

[3] M. F. Sani, S. J. van Zelst, and W. M. P. van der Aalst, "Repairing outlier behaviour in event logs," in Business Information Systems, vol. 320 of Lecture Notes of Business Information Systems, 2018.

[4] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering block-structured process models from event logs - A constructive approach," in Application and Theory of Petri Nets and Concurrency, vol. 7927 of Lecture Notes in Computer Science, pp. 311–329, 2013.
[5] D. Brons, R. Scheepens, and D. Fahland, "Striking a new balance in accuracy and simplicity with the probabilistic inductive miner," in 3rd International Conference on Process Mining, ICPM, 2021.
[6] A. Augusto, R. Conforti, M. Dumas, and M. L. Rosa, "Split miner: Discovering accurate and simple business process models from event logs," in 2017 IEEE International Conference on Data Mining, ICDM 2017, pp. 1–10, IEEE Computer Society, 2017.
[7] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering block-structured process models from event logs containing infrequent behaviour," in Business Process Management Workshops - BPM, vol. 171 of Lecture Notes in Business Information Processing, 2013.
[8] P. Linz, An introduction to formal languages and automata, 4th Edition. Jones and Bartlett Publishers, 2006.
[9] S. L. Bloom and Z. Ésik, "Free shuffle algebras in language varieties," Theor. Comput. Sci., vol. 163, no. 1&2, pp. 55–98, 1996.
[10] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering block-structured process models from incomplete event logs," in Application and Theory of Petri Nets and Concurrency, Lecture Notes in Computer Science, 2014.
[11] P. Fränti and S. Sieranoja, "K-means properties on six clustering benchmark datasets," Appl. Intell., vol. 48, no. 12, pp. 4743–4759, 2018.
[12] E. Verbeek, "Process discovery contest 2022," 2022.
[13] B. van Dongen, "Bpi challenge 2012," 2012.
[14] W. Steeman, "Bpi challenge 2013, incidents," 2013.
[15] W. Steeman, "Bpi challenge 2013, closed problems," 2013.
[16] B. B. van Dongen, "Bpi challenge 2015," 2015.
[17] B. van Dongen, "Bpi challenge 2017," 2017.
[18] B. van Dongen, "Bpi challenge 2019," 2019.
[19] B. van Dongen, "Bpi challenge 2020," 2020.
[20] M. M. de Leoni and F. Mannhardt, "Road traffic fine management process," 2015.
[21] F. Mannhardt, "Sepsis cases - event log," 2016.
[22] B. F. van Dongen, J. Carmona, T. Chatain, and F. Taymouri, "Aligning modeled and observed behavior: A compromise between computation complexity and quality," 2017.
[23] A. Adriansyah, J. Munoz-Gama, J. Carmona, B. F. van Dongen, and W. M. P. van der Aalst, "Measuring precision of modeled behavior," Inf. Syst. E Bus. Manag., vol. 13, no. 1, pp. 37–67, 2015.
[24] A. Augusto, R. Conforti, M. Dumas, M. L. Rosa, F. M. Maggi, A. Marrella, M. Mecella, and A. Soo, "Automated discovery of process models from event logs: Review and benchmark," IEEE Trans. Knowl. Data Eng., vol. 31, no. 4, pp. 686–705, 2019.
[25] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, and G. Bruno, "Automated discovery of structured process models from event logs: The discover-and-structure approach," Data & Knowledge Engineering, vol. 117, pp. 373–392, 2018.
[26] H. M. W. Verbeek and W. M. P. van der Aalst, "Decomposed process mining: The ilp case," in Business Process Management Workshops (F. Fournier and J. Mendling, eds.), (Cham), pp. 264–276, Springer International Publishing, 2015.
[27] R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser, "Process mining based on regions of languages," in Business Process Management (G. Alonso, P. Dadam, and M. Rosemann, eds.), (Berlin, Heidelberg), pp. 375–383, Springer Berlin Heidelberg, 2007.
[28] S. K. vanden Broucke and J. De Weerdt, "Fodina: A robust and flexible heuristic process discovery technique," Decision Support Systems, vol. 100, pp. 109–118, 2017. Smart Business Process Management.
[29] Q. Guo, L. Wen, J. Wang, Z. Yan, and P. S. Yu, "Mining invisible tasks in non-free-choice constructs," in Business Process Management (H. R. Motahari-Nezhad, J. Recker, and M. Weidlich, eds.), (Cham), pp. 109–125, Springer International Publishing, 2015.
[30] A. Weijters, "Process mining: Extending the alpha-algorithm to mine short loops," 06 2004.
[31] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, "Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity," Int. J. Cooperative Inf. Syst., vol. 23, no. 1, 2014.
[32] J. D. Weerdt, S. K. L. M. vanden Broucke, J. Vanthienen, and B. Baesens, "Active trace clustering for improved process discovery," IEEE Trans. Knowl. Data Eng., vol. 25.
[33] B. F. A. Hompes, H. M. W. Verbeek, and W. M. P. van der Aalst, "Finding suitable activity clusters for decomposed process discovery," in Data-Driven Process Discovery and Analysis, vol. 237, 2014.