Discovering Block-Structured Process Models From Event Logs - A Constructive Approach

S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst

Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands

Abstract Process discovery is the problem of, given a log of observed behaviour, finding a process model that 'best' describes this behaviour. A large variety of process discovery algorithms has been proposed. However, no existing algorithm guarantees to return a fitting model (i.e., able to reproduce all observed behaviour) that is sound (free of deadlocks and other anomalies) in finite time. We present an extensible framework to discover from any given log a set of block-structured process models that are sound and fit the observed behaviour. In addition we characterise the minimal information required in the log to rediscover a particular process model. We then provide a polynomial-time algorithm for discovering a sound, fitting, block-structured model from any given log; we give sufficient conditions on the log for which our algorithm returns a model that is language-equivalent to the process model underlying the log, including unseen behaviour. The technique is implemented in a prototypical tool.

Keywords: process discovery, block-structured process models, soundness, fitness

1 Introduction

Process mining techniques aim to extract information from event logs. For example, the audit trails of a workflow management system or the transaction logs of an enterprise resource planning system can be used to discover models describing processes, organisations and products. The most challenging process mining problem is to learn a process model (e.g., a Petri net) from example traces in some event log. Many process discovery techniques have been proposed. For an overview of process discovery algorithms, we refer to [13]. Unfortunately, existing techniques may produce models that are unable to replay the log, may produce erroneous models and may have excessive run times.

Which process model is 'best' is typically defined with respect to several quality criteria. An important quality criterion is soundness. A process model is *sound* if and only if all process steps can be executed and some satisfactory end state is always reachable. In most use cases, an unsound process model can be discarded without considering the log that it should represent. Another model quality criterion is fitness. A model has perfect *fitness* with respect to a log if

it can reproduce all traces in the log. The quality criterion *precision* expresses whether the model does not allow for too much behaviour, *generalisation* expresses that the model will allow future behaviour that is currently absent in the log.[10] Other model quality criteria exist, for which we refer to [23]. In this paper, we focus on soundness and fitness, as so far no existing discovery algorithm guarantees to return a sound fitting model in finite time.

In addition to finite run time, there are other desirable properties of process discovery algorithms. In reality, the log was produced by some real-life process. The original process is *rediscoverable* by a process discovery algorithm if, given a log that contains enough information, the algorithm returns a model that is equivalent to the original process using some equivalence notion. For instance, *language-rediscoverability* holds for an algorithm that returns a model that is language-equivalent to the original model - even if the log obtained from the original model contains less behaviour. *Isomorphic-rediscoverability* holds for an algorithm that returns a model that is isomorphic to (a representation of) the original model. The amount of information that is required to be in the log is referred to as *log completeness*, of which the most extreme case is total log completeness, meaning that all possible behaviour of the original process must be present in the log. A process discovery technique is only useful if it assumes a much weaker notion of completeness. In reality one will rarely see all possible behaviour.

Many process discovery algorithms [5,25,26,24,7,11,27,17,4,19,9,3] using different approaches have been proposed in the past. Some techniques guarantee fitness, e.g., [27], some guarantee soundness, e.g. [9], and others guarantee rediscoverability under some conditions, e.g., [5]. Yet, there is essentially no discovery algorithm guaranteeing to find a sound, fitting model in finite time for all given logs.

In this paper, we use the block-structured process models of [9,3] to introduce a framework that guarantees to return sound and fitting process models. This framework enables us to reason about a variety of quality criteria. The framework uses any flavour of block-structured process models: new blocks/operators can be added without changing the framework and with few proof obligations. The framework uses a divide and conquer approach to decompose the problem of discovering a process model for a log L into discovering n subprocesses of n sublogs obtained by splitting L. We explore the quality standards and hard theoretically founded limits of the framework by characterising the requirements on the log under which the original model can be rediscovered.

For illustrative purposes, we give an algorithm that uses the framework and runs in polynomial time for any log and any number of activities. The framework guarantees that the algorithm returns a sound fitting model. The algorithm works by dividing the activities of the log over a number of branches, such that the log can be split according to this division. We characterise the conditions under which the algorithm returns a model that is language-equivalent to the original process. The algorithm has been prototypically implemented using the ProM framework [12].

In the following, we first discuss related work. Section 3 explains logs, languages, Petri nets, workflow nets and process trees. In Section 4 the framework is described. The class of models that this framework can rediscover is described in Section 5. In Section 6 we give an algorithm that uses the framework and we report on experimental results. We conclude the paper in Section 7.

2 Related work

A multitude of process discovery algorithms has been proposed in the past. We review typical representatives with respect to guarantees such as soundness, fitness, rediscoverability and termination. Techniques that discover process models from ordering relations of activities, such as the α algorithm [5] and its derivatives [25,26], guarantee isomorphic-rediscoverability for rather small classes of models [6] and do not guarantee fitness or soundness. Semantics-based techniques such as the language-based region miner [7,8], the state-based region miner [11], or the ILP miner [27] guarantee fitness but neither soundness nor rediscoverability. Frequency-based techniques such as the heuristics miner [24] guarantee neither soundness nor fitness. Abstraction-based techniques such as the Fuzzy miner [17] produce models that do not have executable semantics and hence guarantee neither soundness nor fitness nor any kind of rediscoverability.

Genetic process discovery algorithms [4,19] may reach certain quality criteria if they are allowed to run forever, but usually cannot guarantee any quality criterion given finite run time. A notable exception is a recent approach [9,3] that guarantees soundness. This approach restricts the search space to block-structured process models, which are sound by construction; however, finding a fitting model cannot be guaranteed in finite run time.

The Refined Process Structure Tree [21] is a parsing technique to find block structures in process models by which soundness can be checked [15], or an arbitrary model can be turned into a block-structured one (if possible) [20]. However, these techniques only analyse or transform a given model, but do not allow to construct a sound or fitting model. The language-based mining technique of [8] uses regular expressions to pre-structure the input language (the log) into smaller blocks; this block-structuring of the log is then used during discovery for constructing a fitting, though possibly unsound, process model.

Unsound models can be repaired to become sound by simulated annealing [16], though fitness to a given log is not preserved. Non-fitting models can be repaired to become fitting by adding subprocesses [14], though soundness is not guaranteed. Hence, a more integrated approach is needed to ensure soundness and fitness. In the following we will propose such an integrated approach building on the ideas of a restriction to block-structured models[3,9], and of decomposing the given log into block-structured parts prior to model construction.

3 Preliminaries

Logs. We assume the set of all process activities Σ to be given. An *event* e is the occurrence of an activity: $e \in \Sigma$. A *trace* t is a possibly empty sequence of events: $t \in \Sigma^*$. We denote the empty trace with ϵ. A *log* L is a finite non-empty set of traces: $L \subseteq \Sigma^*$. For example, $\{\langle a, b, c\rangle, \langle a, c, b\rangle\}$ denotes a log consisting of two traces *abc* and *acb*, where for instance *abc* denotes that first a occurred, then b and finally c. The *size* of a log is the number of events in it: $||L|| = \sum_{t \in L} |t|$.

Petri Nets, Workflow Nets and Block-structured Workflow Nets. A *Petri net* is a bipartite graph containing places and transitions, interconnected by directed arcs. A transition models a process activity, places and arcs model the ordering of process activities. We assume the standard semantics of Petri nets here, see [22]. A *workflow net* is a Petri net having a single start place and a single end place, modeling the start and end state of a process. Moreover, all nodes are on a path from start to end[5]. A *block-structured workflow net* is a hierarchical workflow net that can be divided recursively into parts having single entry and exit points. Figure 1 shows a block-structured workflow net.

Process Trees. A *process tree* is a compact abstract representation of a block-structured workflow net: a rooted tree in which leaves are labeled with activities and all other nodes are labeled with operators. A process tree describes a language, an operator describes how the languages of its subtrees are to be combined.

We formally define process trees recursively. We assume a finite alphabet Σ of activities and a set $\bigoplus$ of operators to be given. Symbol $\tau \notin \Sigma$ denotes the silent activity.

- a with $a \in \Sigma \cup \{\tau\}$ is a process tree;
- Let $M_1, \ldots, M_n$ with $n > 0$ be process trees and let $\oplus$ be a process tree operator, then $\oplus(M_1, \ldots, M_n)$ is a process tree.

There are a few standard operators that we consider in the following: operator $\times$ means the exclusive choice between one of the subtrees, $\rightarrow$ means the sequential execution of all subtrees, $\circlearrowleft$ means the structured loop of loop body M_1 and alternative loop back paths $M_2, \ldots, M_n$, and $\wedge$ means a parallel (interleaved) execution as defined below. Please note that for $\circlearrowleft$, n must be ≥ 2.

To describe the semantics of process trees, we define the language of a process tree M as a recursive monotonic function $\mathcal{L}(M)$, using for each operator $\oplus$ a language join function $\oplus_{\mathrm{l}}$:

$$\begin{aligned} \mathcal{L}(a) &= \{\langle a\rangle\} \text{ for } a \in \Sigma \\ \mathcal{L}(\tau) &= \{\epsilon\} \\ \mathcal{L}(\oplus(M_1, \ldots, M_n)) &= \oplus_{\mathrm{l}}(\mathcal{L}(M_1), \ldots, \mathcal{L}(M_n)) \end{aligned}$$

Each operator $\oplus$ has its own language join function $\oplus_l$. Each function takes several logs and produces a new log: $\oplus_l : 2^{\Sigma^*} \times \ldots \times 2^{\Sigma^*} \rightarrow 2^{\Sigma^*}$.

$$\times_l(L_1, \ldots, L_n) = \bigcup_{1 \leq i \leq n} L_i$$
$$\rightarrow_l(L_1, \ldots, L_n) = \{t_1 \cdot t_2 \cdots t_n | \forall i \in 1 \ldots n : t_i \in L_i\}$$
$$\circlearrowleft_l(L_1, \ldots, L_n) = \{t_1 \cdot t'_1 \cdot t_2 \cdot t'_2 \cdots t_m | \forall i : t_i \in L_1 \wedge t'_i \in \bigcup_{2 \leq j \leq n} L_j\}$$

To characterise $\wedge$, we introduce a set notation $\{t_1, \ldots, t_n\}_{\simeq}$ that interleaves the traces $t_1 \ldots t_n$. We need a more complex notion than a standard projection function due to overlap of activities over traces.

$$t \in \{t_1, \ldots, t_n\}_{\simeq} \Leftrightarrow \exists (f : \{1 \ldots |t|]\} \rightarrow \{(j,k) | j \leq n \wedge k \leq |t_j|\}) :$$
$$\forall i_1 < i_2 \wedge f(i_1) = (j, k_1) \wedge f(i_2) = (j, k_2) : k_1 < k_2 \wedge$$
$$\forall i \leq n \wedge f(i) = (j,k) : t(i) = t_j(k)$$

where f is a bijective function mapping each event of t to an event in one of the t_i and $t(i)$ is the i^{th} element of t. For instance, $\langle a, c, d, b \rangle \in \{\langle a, b \rangle, \langle c, d \rangle\}_{\simeq}$. Using this notation, we define $\wedge_l$:

$$\wedge_l(L_1, \ldots, L_n) = \{t | t \in \{t_1, \ldots, t_n\}_{\simeq} \wedge \forall i : t_i \in L_i\}$$

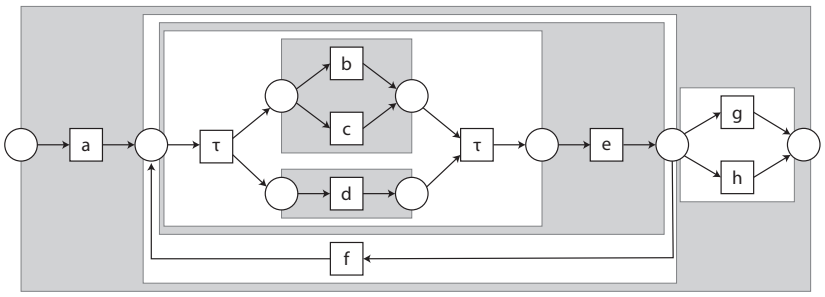


Figure 1: A Petri net, modified from [2, page 196]. The rectangle regions denote the process tree nodes in $\rightarrow(a, \circlearrowleft(\rightarrow(\wedge(\times(b,c),d),e),f),\times(g,h))$.

Each of the process tree operators has a straightforward formal translation to a sound, block-structured workflow Petri net [9,3]. For instance, the Petri net shown in Figure 1 corresponds to the process tree $\rightarrow(a, \circlearrowleft(\rightarrow(\wedge(\times(b,c),d),e),f),$ $\times(g,h))$. If one would come up with another process tree operator, soundness of the translation follows if the translation of the new process tree operator is sound in isolation. The four operators presented here translate to well-structured, free-choice Petri nets [1]; other operators might not.

The *size* of a model M is the number of nodes in M and is denoted as $|M|$: $|\tau| = 1$, $|a| = 1$ and $|\oplus(M_1, \ldots, M_n)| = 1 + \sum_i |M_i|$. Two process trees $M = \oplus(M_1, \ldots, M_n)$ and $M' = \oplus'(M'_1, \ldots, M'_n)$ are *isomorphic* if and only if they are syntactically equivalent up to reordering of children in the case of $\times$, $\wedge$ and the non-first children of $\circlearrowleft$.

If M is a process tree and L is a log, then L *fits* M if and only if every trace in L is in the language of M: $L \subseteq \mathcal{L}(M)$. A *flower model* is a process tree that can produce any sequence of Σ. An example of a flower model is the model $\circlearrowleft(\tau, a_1, \ldots, a_m)$ where $a_1, \ldots, a_m = \Sigma$.

As additional notation, we write $\Sigma(L)$ and $\Sigma(M)$ for the activities occurring in log L or model M respectively, not including τ. Furthermore, $Start(L)$, $Start(M)$ and $End(L)$, $End(M)$ denote the sets of activities with which log L and model M start or end.

4 Framework

In this section, we introduce a highly generic process discovery framework. This framework allows for the derivation of various process discovery algorithms with predefined guarantees. Then we prove that each model returned by the framework fits the log and that the framework describes a finite computation, both for any set of process tree operators having a corresponding monotonic language join function.

Requirement on the Process Tree Operators. The framework works independently of the chosen process tree operators. The only requirement is that each operator $\oplus$ must have a sensible language join function $\oplus_l$, such that the language of $\oplus$ reflects the language join of its $\oplus_l$.

Framework. Given a set $\bigoplus$ of process tree operators, we define a framework B to discover a set of process models using a divide and conquer approach. Given a log L, B searches for possible splits of L into smaller $L_1 \ldots L_n$, such that these logs combined with an operator $\oplus$ can produce L again. It then recurses on the found divisions and returns a cartesian product of the found models. The recursion ends when L cannot be divided any further. We have to give this algorithmic idea a little twist as splitting L into strictly smaller $L_1 \ldots L_n$ could prevent some models from being rediscovered, for instance in presence of unobservable activities. As a more general approach, we allow L to be split into sublogs having the same size as L. However, such splits that do not decrease the size of L may only happen finitely often. For this, we introduce a counter parameter ϕ, which has to decrease if a non-decreasing log split is made. Parameter ϕ essentially bounds the number of invisible branches that the discovered model can have.

The actual split is left to a function $select(L)$ that takes a log and returns a set of tuples $(\oplus, ((L_1, \phi_1), \ldots, (L_n, \phi_n)))$, where $\oplus$ is the operator identified to split L, and $L_1 \ldots L_n$ are the logs obtained by splitting L w.r.t. $\oplus$. Each log L_i has a corresponding counter parameter ϕ which bounds recursion on L_i. Various

select(L) functions can be defined, so we parameterise the framework B with this *select* function.

function $B_{select}(L, \phi)$
  if $L = \{\epsilon\}$ **then**
    $base \leftarrow \{\tau\}$
  else if $\exists a \in \Sigma : L = \{\langle a \rangle\}$ **then**
    $base \leftarrow \{a\}$
  else
    $base \leftarrow \emptyset$
  end if
  $P \leftarrow select(L)$
  if $|P| = 0$ **then**
    if $base = \emptyset$ **then**
      return $\{\circlearrowleft(\tau, a_1, \ldots, a_m)$ where $\{a_1, \ldots, a_m\} = \Sigma(L)\}$
    else
      return $base$
    end if
  end if
  return $\{\oplus(M_1, \ldots, M_n) | (\oplus, ((L_1, \phi_1), \ldots, (L_n, \phi_n))) \in P \wedge \forall i : M_i \in B(L_i, \phi_i)\} \cup base$
end function

Any *select* function can be used, as long as the tuples it returns adhere to the following definition:

Definition 1. *For each tuple* $(\oplus, ((L_1, \phi_1), \ldots, (L_n, \phi_n)))$ *that* $select(L)$ *returns, it must hold that*

$$
\begin{aligned}
& L \subseteq \oplus_{\mathrm{l}}(L_1, \ldots, L_n) \wedge \\
\forall i : & ||L_i|| + \phi_i < ||L|| + \phi \wedge \\
\forall i : & ||L_i|| \leq ||L|| \wedge \\
\forall i : & \phi_i \leq \phi \wedge \\
\forall i : & \Sigma(L_i) \subseteq \Sigma(L) \wedge \\
& \oplus \in \bigoplus \wedge \\
& n \leq ||L|| + \phi
\end{aligned}
$$

In the remainder of this section, we will prove some properties that do not depend on a specific preference function *select*.

Theorem 2. *Assuming select terminates, B terminates.*

Proof. Termination follows from the fact that in each recursion, $||L|| + \phi$ gets strictly smaller and that there are finitely many recursions from a recursion step. By construction of *select*, $\Sigma(L_i) \subseteq \Sigma(L)$, and therefore Σ is finite. By construction of P, $n \leq ||L|| + \phi$, so there are finitely many sublogs L_i. Hence, *select* creates finitely many log divisions. Therefore, the number of recursions is finite and hence B terminates. □

Theorem 3. *Let $\bigoplus$ be a set of operators and let L be a log. Then $B(L)$ returns at least one process tree and all process trees returned by $B(L)$ fit L.*

Proof. Proof by induction on value of $||L|| + \phi$. Base cases: $||L|| + \phi = 1$ or $||L|| + \phi = 2$. Then, L is either $\{\epsilon\}$ or $\{\langle a \rangle\}$. By code inspection, for these L B returns at least one process tree and all process trees fit L.
Induction hypothesis: for all logs $||L'|| + \phi'$ smaller than $||L|| + \phi$, $B(L', \phi')$ returns at least one process tree and all process trees that B returns fit L' : $\forall ||L'|| + \phi' < ||L|| + \phi : |B(L')| \geq 1 \wedge \forall M' \in B(L') : L' \subseteq \mathcal{L}(M')$.
Induction step: assume $||L|| + \phi > 2$ and the induction hypothesis. Four cases apply:

- Case $L = \{\epsilon\}$, see base case;
- Case $L = \{\langle a \rangle\}$, see base case;
- Case P is empty, $L \neq \{\epsilon\}$ and $L \neq \{a\}$. Then B returns the flower model $\{\circlearrowleft(\tau, a_1, \ldots, a_m)$ where $a_1, \ldots, a_m = \Sigma(L)\}$ and that fits any log.
- Case P is nonempty, $L \neq \{\epsilon\}$ and $L \neq \{a\}$. Let $M_1, \ldots, M_n$ be models returned by $B(L_1, \phi_1), \ldots, B(L_n, \phi_n)$ for some logs $L_1, \ldots, L_n$ and some counters $\phi_1, \ldots, \phi_n$. By construction of the P-selection step, $\forall i : ||L_i|| + \phi_i < ||L|| + \phi$. By the induction hypothesis, these models exist. As B combines these models in a cartesian product, $|B(L)| \geq 1$. By the induction hypothesis, $\forall i : L_i \subseteq \mathcal{L}(M_i)$. Using the fact that $\oplus_l$ is monotonic and the construction of M, we obtain $\oplus_l(L_1, \ldots, L_n) \subseteq \mathcal{L}(\oplus(M_1, \ldots, M_n)) = \mathcal{L}(M)$. By construction of P, $L \subseteq \oplus_l(L_1, \ldots, L_n)$, and by $\oplus_l(L_1, \ldots, L_n) \subseteq L(M)$, we conclude that $L \subseteq \mathcal{L}(M)$. We did not pose any restrictions on $M_1, \ldots, M_n$, so this holds for all combinations of $M_1, \ldots, M_n$ from the sets returned by B. □

5 Rediscoverability of Process Trees

An interesting property of a discovery algorithm is whether and under which assumptions an original process can be rediscovered by the algorithm. Assume the original process is expressible as a model M, which is unknown to us. Given is a log L of M: $L \subseteq \mathcal{L}(M)$. M is isomorphic-rediscoverable from L by algorithm B if and only if $M \in B(L)$. It is desirable that L can be as small as possible to rediscover M. In this section, we explore the boundaries of the framework B of Section 4 in terms of rediscoverability.

We first informally give the class of original processes that can be rediscovered by B, and assumptions on the log under which this is guaranteed. After that, we give an idea why these suffice to rediscover the model. In this section, the preference function *select* as used in B is assumed to be the function returning all log divisions satisfying Definition 1. Otherwise, the original model might be removed from the result set.

Class of Rediscoverable Models. Any algorithm has a representational bias; B can only rediscover processes that can be described by process trees. There are no further limitations: B can rediscover every process tree. An intuitive argument

for this claim is that as long as the log can be split into the parts of which the log was constructed, the algorithm will also make this split and recurse. A necessity for this is that the log contains 'enough' behaviour.

Log Requirements. All process trees can be rediscovered given 'enough' traces in the log, where enough means that the given log can be split according to the respective process tree operator. Intuitively, it suffices to execute each occurrence of each activity in M at least once in L. Given a large enough ϕ, B can then always split the log correctly.

This yields the notion of *activity-completeness*. Log L is activity-complete w.r.t. model M, denote $L \diamond_{\mathrm{a}} M$, if and only if each leaf of M appears in L at least once. Formally, we have to distinguish two cases. For a model M' where each activity occurs at most once and a log L',

$$L' \diamond_{\mathrm{a}} M' \Leftrightarrow \Sigma(M') \subseteq \Sigma(L')$$

In the general case, where some activity $a \in \Sigma$ occurs more than once in M, we have to distinguish the different occurrences. For a given alphabet Σ consider a refined alphabet Σ' and a surjective function $f : \Sigma' \to \Sigma$, e.g., $\Sigma' = \{a_1, a_2, \ldots, b_1, b_2, \ldots\}$ and $a = f(a_1) = f(a_2) = \cdots$, $b = f(b_1) = f(b_2) = \cdots$, etc. For a log L' and model M' over Σ', let $f(L')$ and $f(M')$ denote the log and the model obtained by replacing each $a \in \Sigma'$ by $f(a) \in \Sigma$. Using this notation, we define for arbitrary log L and model M,

$$L \diamond_{\mathrm{a}} M \Leftrightarrow \exists \Sigma', (f' : \Sigma' \to \Sigma), M', L' : f(L') = L \wedge f(M') = M \wedge L' \diamond_{\mathrm{a}} M',$$

where each activity $a \in \Sigma'$ occurs at most once in M'.

Rediscoverability of Models. In order to prove isomorphic rediscoverability, we need to show that any log $L \diamond_{\mathrm{a}} M$ can be split by B such that M can be constructed, given a large enough ϕ.

Theorem 4. *Given a large enough ϕ, for each log L and model M such that $L \subseteq \mathcal{L}(M)$ and $L \diamond_{\mathrm{a}} M$ it holds that $M \in B(L)$.*

Proof. Proof by induction on model sizes. Base case: $|M| = 1$. A model of size 1 consists of a single leaf l. By $L \subseteq \mathcal{L}(M) \wedge L \diamond_{\mathrm{a}} M$, L is $\{l\}$. These are handled by the $L = \{\epsilon\}$ or $L = \{\langle a \rangle\}$ clauses and hence can be rediscovered.
Induction hypothesis: all models smaller than M can be rediscovered: $\forall |M'| < |M| \wedge L' \subseteq \mathcal{L}(M) \wedge L' \diamond_{\mathrm{a}} M' : M' \in B(L', \phi')$, for some number ϕ'.
Induction step: assume $|M| > 1$ and the induction hypothesis. As $|M| > 1$, $M = \oplus(M_1, \ldots, M_n)$ for certain $\oplus$, n and $M_1 \ldots M_n$. By $L \subseteq \mathcal{L}(M)$ and definition of $\mathcal{L}(M)$, there exist $L_1 \ldots L_n$ such that $\forall i : L_i \subseteq \mathcal{L}(M_i)$, $\forall i : L_i \diamond_{\mathrm{a}} M_i$ and $L \subseteq \oplus_{\mathrm{l}}(L_1, \ldots, L_n)$. By the induction hypothesis there exist $\phi_1 \ldots \phi_n$ such that $\forall i : M_i \in B(L_i, \phi_i)$. We choose ϕ to be large enough by taking $\phi = \max\{n, \phi_1 + 1, \ldots, \phi_n + 1\}$. By this choice of ϕ,

$$\forall i : ||L_i|| + \phi_i < ||L|| + \phi \wedge \phi_i \leq \phi$$

and

$$n \leq ||L|| + \phi$$

hold. By construction of $\oplus_l$,

$$\forall i : ||L_i|| \leq ||L||$$

By $|M| > 1$ and our definitions of $\times_l$, $\rightarrow_l$, $\wedge_l$ and $\circlearrowleft_l$, L does not introduce new activities:

$$\forall i : \Sigma(L_i) \subseteq \Sigma(L)$$

Hence, $(\oplus, ((L_1, \phi_1), \ldots, (L_n, \phi_n))) \in P$. By the induction hypothesis, $\forall M_i : M_i \in B(L_i)$. The models returned by $B(L_i)$ will be combined using a cartesian product, and as $M = \oplus(M_1, \ldots, M_n)$, it holds that $M \in B(L)$. □

This proof shows that it suffices to pick ϕ to be the sum of the width and depth of the original model M in order to rediscover M from an activity-complete log L.

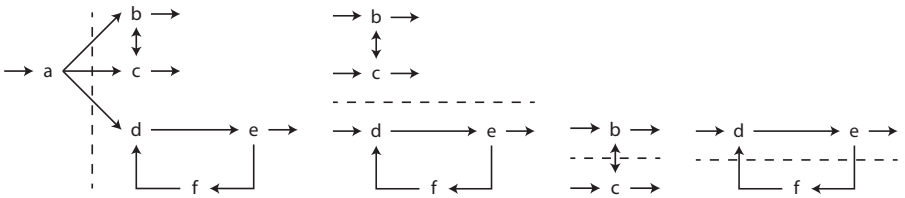
6 Discovering Process Trees Efficiently

The framework of Section 4 has a practical limitation: for most real-life logs, it is infeasible to construct the full set P. In this section, we introduce an algorithm B' that is a refinement of the framework B. B' avoids constructing the complete set P. The central idea of B' is to compute a log split directly based on the ordering of activities in the log. We first introduce the algorithmic idea and provide formal definitions afterwards. We conclude this section with a description of the classes of process trees that B' is able to language-rediscover and a description of our prototype implementation.

6.1 Algorithmic idea

The directly-follows relation, also used by the α-algorithm [5], describes when two activities directly follow each other in a process. This relation can be expressed in the *directly-follows graph* of a log L, written $G(L)$. It is a directed graph containing as nodes the activities of L. An edge (a, b) is present in $G(L)$ if and only if some trace $\langle \cdots, a, b, \cdots \rangle$ exists in L. A node of $G(L)$ is a *start node* if its activity is in $Start(L)$. We define $Start(G(L)) = Start(L)$. Similarly for end nodes in $End(L)$, and $End(G(L))$. The definition for $G(M)$ is similar. For instance, Figure 2a shows the directly-follows graph of log $L = \{\langle a, b, c\rangle, \langle a, c, b\rangle, \langle a, d, e\rangle, \langle a, d, e, f, d, e\rangle\}$.

The idea for our algorithm is to find in $G(L)$ structures that indicate the 'dominant' operator that orders the behaviour. For example, $G(L)$ of Fig 2a can be partitioned into two sets of activities as indicated by the dashed line such that edges cross the line only from left to right. This pattern corresponds to a sequence where the activities left of the line precede the activities right of the line. This is the decisive hint on how to split a given log when using the framework of



(a) $G(L)$. (b) $G(L_2)$ (c) $G(L_3)$. (d) $G(L_4)$.

Figure 2: Several directly-follows graphs. Dashed lines denote cuts.

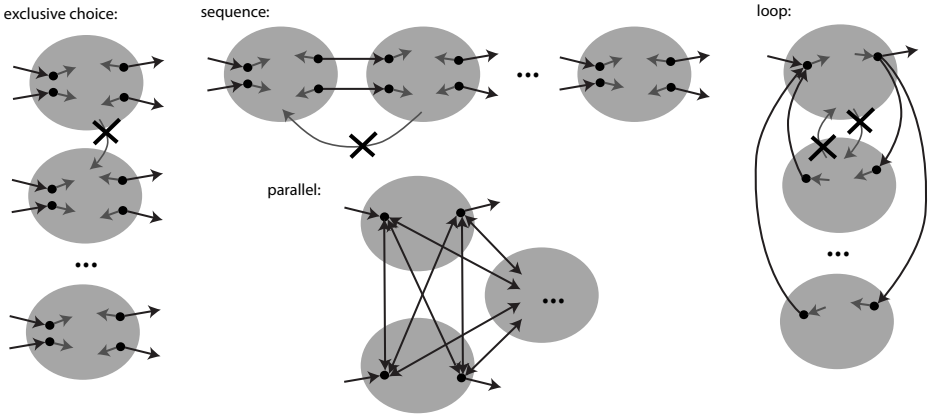


Figure 3: Cuts of the directly-follows graph for operators $\times$, $\rightarrow$, $\wedge$ and $\circlearrowleft$.

Section 4. Each of the four operators $\times$, $\rightarrow$, $\circlearrowleft$, $\wedge$ has a characteristic pattern in $G(L)$ that can be identified by finding a partitioning of the nodes of $G(L)$ into n sets of nodes with characteristic edges in between. The log L can then be split according to the identified operator, and the framework recurses on each of the split logs. The formal definitions are provided next.

6.2 Cuts and Components

Let $G(L)$ be the directly-follows graph of a log L. An *n-ary cut* c of $G(L)$ is a partition of the nodes of the graph into disjoint sets $\Sigma_1 \dots \Sigma_n$. We characterise a different cut for each operator $\times$, $\rightarrow$, $\circlearrowleft$, $\wedge$ based on edges between the nodes.

In a *exclusive choice cut*, each Σ_i has a start node and an end node, and there is no edge between two different $\Sigma_i \neq \Sigma_j$, as illustrated by Figure 3(left). In a *sequence cut*, the sets $\Sigma_1 \dots \Sigma_n$ are ordered such that for any two nodes $a \in \Sigma_i, b \in \Sigma_j, i < j$, there is a path from a to b along the edges of $G(L)$, but not vice versa; see Figure 3(top). In a *parallel cut*, each Σ_i has a start node and an end node, and any two nodes $a \in \Sigma_i, b \in \Sigma_j, i \neq j$ are connected by edges (a, b) and (b, a); see Figure 3(bottom). In a *loop cut*, Σ_1 has all start and all end

nodes of $G(L)$, there is no edge between nodes of different $\Sigma_i \neq \Sigma_j, i, j > 1$, and any edge between Σ_1 and $\Sigma_i, i > 1$ either leaves an end node of Σ_1 or reaches a start node of Σ_1; see Figure 3(right). An n-ary cut is *maximal* if there exists no cut of G of which n is bigger. A cut c is *nontrivial* if $n > 1$.

Let $a \rightsquigarrow b \in G$ denote that there exists a directed edge chain (path) from a to b in G. Definitions 5, 6, 7 and 8 show the formal cut definitions.

Definition 5. *An* exclusive choice cut *is a cut $\Sigma_1, \ldots, \Sigma_n$ of a directly-follows graph G, such that*

1. $\forall i \neq j \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : (a_i, a_j) \notin G$

Definition 6. *A* sequence cut *is an ordered cut $\Sigma_1, \ldots, \Sigma_n$ of a directly-follows graph G such that*

1. $\forall 1 \leq i < j \leq n \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : a_j \rightsquigarrow a_i \notin G$
2. $\forall 1 \leq i < j \leq n \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : a_i \rightsquigarrow a_j \in G$

Definition 7. *A* parallel cut *is a cut $\Sigma_1, \ldots, \Sigma_n$ of a directly-follows graph G such that*

1. $\forall i : \Sigma_i \cap Start(G) \neq \emptyset \wedge \Sigma_i \cap End(G) \neq \emptyset$
2. $\forall i \neq j \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : (a_i, a_j) \in G \wedge (a_j, a_i) \in G$

Definition 8. *A* loop cut *is a partially ordered cut $\Sigma_1, \ldots, \Sigma_n$ of a directly-follows graph G such that*

1. $Start(G) \cup End(G) \subseteq \Sigma_1$
2. $\forall i \neq 1 \wedge a_i \in \Sigma_i \wedge a_1 \in \Sigma_1 : (a_1, a_i) \in G \Rightarrow a_1 \in End(G)$
3. $\forall i \neq 1 \wedge a_i \in \Sigma_i \wedge a_1 \in \Sigma_1 : (a_i, a_1) \in G \Rightarrow a_1 \in Start(G)$
4. $\forall 1 \neq i \neq j \neq 1 \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : (a_i, a_j) \notin G$
5. $\forall i \neq 1 \wedge a_i \in \Sigma_i \wedge a_1 \in Start(G) : (\exists a'_1 \in \Sigma_1 : (a_i, a'_1) \in G) \Leftrightarrow (a_i, a_1) \in G$
6. $\forall i \neq 1 \wedge a_i \in \Sigma_i \wedge a_1 \in End(G) : (\exists a'_1 \in \Sigma_1 : (a'_1, a_i) \in G) \Leftrightarrow (a_1, a_i) \in G$

6.3 Algorithm B'

B' uses the framework B of Section 4 by providing a *select* function $select_{B'}$. $select_{B'}$ takes a log and produces a single log division. Recursion and base cases are still handled by the framework B. For ϕ, we will use the fixed value 0.

The function $select_{B'}$ works by constructing the directly follows graph $G(L)$ of the input log L. After that, the function tries to find one of the four cuts characterised above. If $select_{B'}$ finds such a cut, it splits the log according to the cut and returns a log division corresponding to the cut. If $select_{B'}$ cannot find a cut, it returns no log division, and B will produce the flower model for L.

We first define $select_{B'}$ followed by the functions to split the log which we illustrate by a running example. We conclude by posing a lemma stating that $select_{B'}$ is a valid *select* function for the framework B.

function $select_{B'}(L)$

if $\epsilon \in L \vee \exists a \in \Sigma(L) : L = \{\langle a \rangle\}$ **then**
return $\emptyset$
else if $c \leftarrow$ a nontrivial maximal exclusive choice cut c of $G(L)$ **then**
$\Sigma_1, \ldots, \Sigma_n \leftarrow c$
$L_1, \ldots, L_n \leftarrow$ ExclusiveChoiceSplit$(L, (\Sigma_1, \ldots, \Sigma_n))$
return $\{(\times, ((L_1, 0), \ldots, (L_n, 0)))\}$
else if $c \leftarrow$ a nontrivial maximal sequence cut c of $G(L)$ **then**
$\Sigma_1, \ldots, \Sigma_n \leftarrow c$
$L_1, \ldots, L_n \leftarrow$ SequenceSplit$(L, (\Sigma_1, \ldots, \Sigma_n))$
return $\{(\rightarrow, ((L_1, 0), \ldots, (L_n, 0)))\}$
else if $c \leftarrow$ a nontrivial maximal parallel cut c of $G(L)$ **then**
$\Sigma_1, \ldots, \Sigma_n \leftarrow c$
$L_1, \ldots, L_n \leftarrow$ ParallelSplit$(L, (\Sigma_1, \ldots, \Sigma_n))$
return $\{(\wedge, ((L_1, 0), \ldots, (L_n, 0)))\}$
else if $c \leftarrow$ a nontrivial maximal loop cut c of $G(L)$ **then**
$\Sigma_1, \ldots, \Sigma_n \leftarrow c$
$L_1, \ldots, L_n \leftarrow$ LoopSplit$(L, (\Sigma_1, \ldots, \Sigma_n))$
return $\{(\circlearrowleft, ((L_1, 0), \ldots, (L_n, 0)))\}$
end if
return $\emptyset$
end function

Using the cut definitions, $select_{B'}$ divides the activities into sets $\Sigma_1 \ldots \Sigma_n$. After that, $select_{B'}$ splits the log.

The cuts can be computed efficiently using graph techniques. We will give an intuition: the exclusive choice cut corresponds to the notion of connected components. If we collapse both strongly connected components and pairwise unreachable nodes into single nodes, the collapsed nodes that are left are the Σs of the sequence cut. If both of these cuts are not present, then we "invert" the graph by removing every double edge, and adding double edges where there was no or a single edge present. In the resulting graph each connected component is a Σ_i of the parallel cut. If these cuts are not present, temporarily removing the start and end activities and computing the connected components in the resulting graph roughly gives us the loop cut. As shown in Lemma 16 in [18], the order in which the cuts are searched for is arbitrary, but for ease of proof and computation we assume it to be fixed as described in $select_{B'}$.

We define the log split functions together with a running example.

Consider the log $L = \{\langle a, b, c \rangle, \langle a, c, b \rangle, \langle a, d, e \rangle, \langle a, d, e, f, d, e \rangle\}$. $G(L)$ is shown in Figure 2a which has the sequence cut $\{a\}, \{b, c, d, e, f\}$. The log is then split by projecting each trace of L onto the different activity sets of the cut.

function SequenceSplit$(L, (\Sigma_1, \ldots, \Sigma_n))$
$\forall j : L_j \leftarrow \{t_j | t_1 \cdot t_2 \cdots t_n \in L \wedge \forall i \leq n \wedge e \in t_i : e \in \Sigma_i\}$
return $L_1, \ldots, L_n$
end function

In the example, SequenceSplit$(L, (\{a\}, \{b, c, d, e, f\})) =$
$\{\langle a \rangle\}, \{\langle b, c \rangle, \langle c, b \rangle, \langle d, e \rangle, \langle d, e, f, d, e \rangle\}$. Call the second log L_2. $G(L_2)$ is shown

in Figure 2b and has the exclusive choice cut $\{b, c\}, \{d, e, f\}$. The log is then split by moving each trace of L into the log of the corresponding activity set.

function ExclusiveChoiceSplit$(L, (\Sigma_1, \ldots, \Sigma_n))$
  $\forall i : L_i \leftarrow \{t | t \in L \wedge \forall e \in t : e \in \Sigma_i\}$
  return $L_1, \ldots, L_n$
end function

In the example, ExclusiveChoiceSplit$(L_2, (\{b, c\}, \{d, e, f\})) = \{\langle b, c\rangle, \langle c, b\rangle\}$, $\{\langle d, e\rangle, \langle d, e, f, d, e\rangle\}$. Call the first log L_3 and the second log L_4. $G(L_3)$ is shown in Figure 2c and has the the parallel cut $\{b\}, \{c\}$. The log is split by projecting each trace for each activity set in the cut.

function ParallelSplit$(L, (\Sigma_1, \ldots, \Sigma_n))$
  $\forall i : L_i \leftarrow \{t|_{\Sigma_j} | t \in L\}$
  return $L_1, \ldots, L_n$
end function

where $t|_X$ is a function that projects trace t onto set of activities X, such that all events remaining in $t|_X$ are in X. In our example, ParallelSplit$(L_3, (\{b\}, \{c\})) = \{\langle b\rangle\}, \{\langle c\rangle\}$. The directly-follows graph of the log $L_4 = \{\langle d, e\rangle, \langle d, e, f, d, e\rangle\}$ is shown in Figure 2d and has the loop cut $\{d, e\}, \{f\}$. The log is split by splitting each trace into subtraces of the loop body and of the loopback condition which are then added to the respective sublogs.

function LoopSplit$(L, (\Sigma_1, \ldots, \Sigma_n))$
  $\forall i : L_i \leftarrow \{t_2 | t_1 \cdot t_2 \cdot t_3 \in L \wedge$
    $\Sigma(\{t_2\}) \subseteq \Sigma_i \wedge$
    $(t_1 = \epsilon \vee (t_1 = \langle \cdots, a_1\rangle \wedge a_1 \notin \Sigma_i)) \wedge$
    $(t_3 = \epsilon \vee (t_3 = \langle a_3, \cdots\rangle \wedge a_3 \notin \Sigma_i))\}$
  return $L_1, \ldots, L_n$
end function

In our example, LoopSplit$(L_4, (\{d, e\}, \{f\})) = \{\langle d, e\rangle\}, \{\langle f\rangle\}$.

Framework B and $select_{B'}$ together discover a process model from the log $L = \{\langle a, b, c\rangle, \langle a, c, b\rangle, \langle a, d, e\rangle, \langle a, d, e, f, d, e\rangle\}$ as follows. The only exclusive choice cut for $G(L)$ is $\{a\}, \{b, c, d, e, f\}$, which is a trivial cut. As we have shown before, a sequence cut for $G(L)$ is $\{a\}, \{b, c, d, e, f\}$. Then, $select_{B'}$ calls SequenceSplit, which returns two sublogs: $L_1 = \{\langle a\rangle\}$ and $L_2 = \{\langle b, c\rangle, \langle c, b\rangle, \langle d, e\rangle, \langle d, e, f, d, e\rangle\}$. Then, $select_{B'}$ returns $\{\rightarrow, (L_1, L_2)\}$. After that, B constructs the partial model $M = \rightarrow(B(L_1), B(L_2))$ and recurses.

Let us first process the log L_1. $B(L_1)$ sets *base* to $\{a\}$, and $select_{B'}(L_1)$ returns $\emptyset$. Then, B returns the process tree a, with which the partially discovered model becomes $M = \rightarrow(a, B(L_2))$.

For $B(L_2)$, ExclusiveChoiceSplit splits the log in $L_3 = \{\langle b, c\rangle, \langle c, b\rangle\}$ and $L_4 = \{\langle d, e\rangle, \langle d, e, f, d, e\rangle\}$. The partially discovered model then becomes $M = \rightarrow(a, \times(B(L_3), B(L_4)))$.

For $B(L_3)$, there is no nontrivial exclusive choice cut and neither a nontrivial sequence cut. As we have shown before, ParallelSplit splits L_3 into $L_5 = \{\langle b \rangle\}$ and $L_6 = \{\langle c \rangle\}$. M becomes $\rightarrow(a, \times(\wedge(B(L_5), B(L_6)), B(L_4)))$.

For $B(L_4)$, LoopSplit splits L_4 into $L_7 = \{\langle d, e \rangle\}$ and $L_8 = \{\langle f \rangle\}$, such that M becomes $\rightarrow(a, \times(\wedge(B(L_5), B(L_6)), \circlearrowleft(L_7, L_8)))$.

After one more sequence cut ($B(L_7)$) and a few base cases ($B(L_5), B(L_6), B(L_7)$), B' discovers the model $\rightarrow(a, \times(\wedge(b, c), \circlearrowleft(\rightarrow(d, e), f)))$.

As B' uses a *select* function to use the framework, we need to prove that $select_{B'}$ only produces log divisions that satisfy Definition 1.

Lemma 9. *The log divisions of L that $select_{B'}$ returns adhere to Definition 1.*

The proof idea of this lemma is to show that each of the clauses of Definition 1 holds for the log division $L_1 \dots L_n$ that $select_{B'}$ chooses, using a fixed ϕ of 0: $L \subseteq \oplus_1(L_1, \dots, L_n)$, $\forall i : ||L_i|| < ||L||$, $\forall i : \Sigma(L_i) \subseteq \Sigma(L)$, $\oplus \in \bigoplus$ and $n \leq ||L||$. For the detailed proof of this lemma, please refer to [18].

6.4 Language-rediscoverability

An interesting property of a discovery algorithm is whether and under which assumptions a model can be discovered that is language-equivalent to the original process. It can easily be inductively proven that B' returns a single process tree for any log L. B' language-rediscovers a process model if and only if the mined process model is language-equivalent to the original process model that produced the log: $\mathcal{L}(M) = \mathcal{L}(B'(L))$ (we abuse notation a bit here), under the assumption that L is complete w.r.t. M for some completeness notion. Our proof strategy for language-rediscoverability will be to reduce each process tree to a normal form and then prove that B' isomorphically rediscovers this normal form. We first define the log completeness notion, after which we describe the class of models that B' can language-rediscover. We conclude with a definition of the normal form and the proof.

Log Completeness. Earlier, we introduced the notion of a directly-follows graph. This yields the notion of *directly-follows completeness* of a log L with respect to a model M, written as $L \diamond_{\text{df}} M$: $L \diamond_{\text{df}} M \equiv \langle \cdots, a, b, \cdots \rangle \in \mathcal{L}(M) \Rightarrow \langle \cdots, a, b, \cdots \rangle \in L \wedge Start(M) \subseteq Start(L) \wedge End(M) \subseteq End(L) \wedge \Sigma(M) \subseteq \Sigma(L)$. Intuitively, the directly-follows graphs M must be mappable on the directly-follows graph of L.

Please note that the framework does not require the log to be directly-follows complete in order to guarantee soundness and fitness.

Class of Language-rediscoverable Models. Given a model M and a generated complete log L, we prove language-rediscoverability assuming the following model restrictions, where $\oplus(M_1, \dots, M_n)$ is a node at any position in M:

1. Duplicate activities are not allowed: $\forall i \neq j : \Sigma(M_i) \cap \Sigma(M_j) = \emptyset$.

2. If $\oplus = \circlearrowleft$, the sets of start and end activities of the first branch must be disjoint: $\oplus = \circlearrowleft \Rightarrow Start(M_1) \cap End(M_1) = \emptyset$.
3. No τ's are allowed: $\forall i \leq n : M_i \neq \tau$.

A reader familiar with the matter will have recognised the restrictions as similar to the rediscoverability restrictions of the α algorithm [5].

Normal form. We first introduce reduction rules on process trees that transform an arbitrary process tree into a normal form. The intuitive idea of these rules is to combine multiple nested subtrees with the same operator into one node with that operator.

Property 10.

$$\begin{aligned}
\oplus(M) &= M \\
\times(\cdots_1, \times(\cdots_2), \cdots_3) &= \times(\cdots_1, \cdots_2, \cdots_3) \\
\rightarrow(\cdots_1, \rightarrow(\cdots_2), \cdots_3) &= \rightarrow(\cdots_1, \cdots_2, \cdots_3) \\
\wedge(\cdots_1, \wedge(\cdots_2), \cdots_3) &= \wedge(\cdots_1, \cdots_2, \cdots_3) \\
\circlearrowleft(\circlearrowleft(M, \cdots_1), \cdots_2) &= \circlearrowleft(M, \cdots_1, \cdots_2) \\
\circlearrowleft(M, \cdots_1, \times(\cdots_2), \cdots_3) &= \circlearrowleft(M, \cdots_1, \cdots_2, \cdots_3)
\end{aligned}$$

It is not hard to reason that these rules preserve language. A process tree on which these rules have been applied exhaustively is a *reduced* process tree. For a reduced process tree it holds that a) for all nodes $\oplus(M_1, \ldots, M_n)$, $n > 1$; b) $\times$, $\rightarrow$ and $\wedge$ do not have a direct child of the same operator; and c) the first child of a $\circlearrowleft$ is not a $\circlearrowleft$ and any non-first child is not an $\times$.

Language-rediscoverability. Our proof strategy is to first exhaustively reduce the given model M to some language-equivalent model M'. After that, we prove that B' discovers M' isomorphically. We use two lemmas to prove that a directly-follows complete log in each step always only allows to 1) pick one specific process tree operator, and 2) split the log in one particular way so that M' is inevitably rediscovered.

Lemma 11. *Let $M = \oplus(M_1, \ldots, M_n)$ be a reduced model that adheres to the model restrictions and let L be a log such that $L \diamond_{\mathrm{df}} M$. Then $select_{B'}$ selects $\oplus$.*

The proof strategy is to prove for all operators that given a directly-follows complete log L of some process tree $\oplus(\ldots)$, $\oplus$ will be the first operator for which $G(L)$ satisfies all cut criteria according to the order $\times$, $\rightarrow$, $\wedge$, $\circlearrowleft$, which is the order in which the operators are checked in $select_{B'}$. For instance, for $\times$, $G(L)$ cannot be connected and therefore will B will select $\times$. For more details, see [18].

Lemma 12. *For each reduced process tree $M = a$ (with a in Σ) or $M = \tau$, and a log L that fits and is directly-follows complete to M, it holds that $M = B'(L)$.*

This lemma is proven by a case distinction on M being either τ or $a \in \Sigma$, and code inspection of $select_{B'}$. For more details, see [18].

Lemma 13. *Let $M = \oplus(M_1, \ldots, M_n)$ be a reduced process tree adhering to the model restrictions, and let L be a log such that $L \subseteq \mathcal{L}(M) \wedge L \diamond_{\mathrm{df}} M$. Let $\{(\oplus, ((L_1, 0), \ldots, (L_n, 0)))\}$ be the result of $select_{B'}$. Then $\forall i : L_i \subseteq \mathcal{L}(M_i) \wedge L_i \diamond_{\mathrm{df}} M_i$.*

The proof strategy is to show for each operator $\oplus$, that the cut $\Sigma_1 \ldots \Sigma_n$ that $select_{B'}$ chooses is the correct activity division ($\forall i : \Sigma_i = \Sigma(M_i) = \Sigma(L_i)$). Using that division, we prove that the Split function returns sublogs $L_1 \ldots L_n$ that are valid for their submodels ($\forall i : L_i \subseteq \mathcal{L}(M_i)$). We then show that each sublog produced by Split produces a log that is directly-follows complete w.r.t. its submodel ($\forall i : L_i \diamond_{\mathrm{df}} M_i$). See [18] for details.

Using these lemmas, we prove language-rediscoverability.

Theorem 14. *If the model restrictions hold for a process tree M, then B' language-rediscovers M: $\mathcal{L}(M) = \mathcal{L}(B'(L))$ for any log L such that $L \subseteq \mathcal{L}(M) \wedge L \diamond_{\mathrm{df}} M$.*

We prove this theorem by showing that a reduced version M' of M is isomorphic to the model returned by B', which we prove by induction on model sizes. Lemma 12 proves isomorphism of the base cases. In the induction step, Lemma 11 ensures that $B'(L)$ has the same root operator as M, and Lemma 13 ensures that the subtrees of M' are isomorphically rediscovered as subtrees of $B'(L)$. For a detailed proof see [18].

Corollary 15. *The process tree reduction rules given in Property 10 yield a language-unique normal form.*

Take a model M that adheres to the model restrictions. Let $L \subseteq \mathcal{L}(M) \wedge L \diamond_{\mathrm{df}} M$ and $M' = B'(L)$. Let M'' be another model adhering to the model restrictions and fitting L. As proven in Lemma 16 in [18], the cuts the algorithm took are mutually exclusive. That means that at each position in the tree, only two options exist that lead to fitness: either the operator $\oplus \in \{\times, \rightarrow, \wedge, \circlearrowleft\}$ chosen by B', or a flower model. By Theorem 14, $B'(L)$ never chose the flower model. Therefore, $B'(L)$ returns the most-precise fitting process tree adhering to the model restrictions. According to the definitions in [10], M' is a model of perfect simplicity and generalisation: M' contains no duplicate activities (simplicity) and any trace that can be produced by M in the future can also be produced by M' (generalisation). By Corollary 15 and construction of Property 10, it is the smallest process tree model having the same language as well.

6.5 Tool support

We implemented a prototype of the B' algorithm as the InductiveMiner plugin of the ProM framework [12], see http://www.promtools.org/prom6/. Here, we sketch its run time complexity and illustrate it with a mined log.

Run Time Complexity. We sketch how we implemented B' as a polynomial algorithm. Given a log L, B' returns a tree in which each activity occurs once, each call of B' returns one tree, and B' recurses on each node once, so the number of recursions is $O(|\Sigma(L)|)$. In each recursion, B' traverses the log and searches for a graph cut. In Section 6.2, we sketched how directly-follows graph cuts can be found using standard (strongly) connected components computations. The exclusive choice, parallel and loop cuts were translated to finding connected components, the sequence cut to finding strongly connected components. For these common graph problems, polynomial algorithms exist. B' is implemented as a polynomial algorithm.

Illustrative Result. To illustrate our prototype, we fed it a log, obtained from [2, page 195]: $L = \{\langle a, c, d, e, h\rangle, \langle a, b, d, e, g\rangle, \langle a, d, c, e, h\rangle, \langle a, b, d, e, h\rangle, \langle a, c, d, e, g\rangle,$

$\langle a, d, c, e, g\rangle, \langle a, b, d, e, h\rangle, \langle a, c, d, e, f, d, b, e, h\rangle, \langle a, d, b, e, g\rangle, \langle a, c, d, e, f, b, d, e, h\rangle,$
$\langle a, c, d, e, f, b, d, e, g\rangle, \langle a, c, d, e, f, d, b, e, g\rangle, \langle a, d, c, e, f, c, d, e, h\rangle,$
$\langle a, d, c, e, f, d, b, e, h\rangle, \langle a, d, c, e, f, b, d, e, g\rangle, \langle a, c, d, e, f, b, d, e, f, d, b, e, g\rangle,$
$\langle a, d, c, e, f, d, b, e, g\rangle, \langle a, d, c, e, f, b, d, e, f, b, d, e, g\rangle, \langle a, d, c, e, f, d, b, e, f, b, d, e, h\rangle,$
$\langle a, d, b, e, f, b, d, e, f, d, b, e, g\rangle, \langle a, d, c, e, f, d, b, e, f, c, d, e, f, d, b, e, g\rangle\}$. The result of our implementation is $M' = \rightarrow(a, \circlearrowleft(\rightarrow(\wedge(\times(b, c), d), e), f), \times(h, g))$. A manual inspection reveals that this model indeed fits the log.

Take an arbitrary model M that could have produced L such that L is directly-follows complete w.r.t. M. Then by Theorem 14, $\mathcal{L}(M) = \mathcal{L}(M')$.

7 Conclusion

Existing process discovery techniques cannot guarantee soundness, fitness, rediscoverability and finite run time at the same time. We presented a process discovery framework B and proved that B produces a set of sound, fitting models in finite time. We described the conditions on the process tree operators under which the framework achieves this. The process tree operators $\times$, $\rightarrow$, $\wedge$ and $\circlearrowleft$ satisfy these conditions. However, the framework is extensible and could be applied to other operators, provided these satisfy the conditions. Another way of positioning our work is that our approach is able to discover some τ transitions in models for which the α-algorithm fails.

To make the framework even more extensible, it uses a to-be-given preference function *select* that selects preferred log divisions. Soundness, fitness and framework termination are guaranteed for any *select* adhering to B. We showed that if the model underlying the log is a process tree, then B can isomorphically-rediscover the model.

To illustrate B, we introduced an algorithm B' that uses B and returns a single process tree. B' works by dividing the activities in the log into sets, after which it splits the log over those sets. We proved that $select_{B'}$ adheres to B, which guarantees us soundness, fitness and framework termination for any input log. We proved that if the model underlying the log is representable as a process

tree that has no duplicate activities, contains no silent activities and where all loop bodies contain no activity that is both a start and end activity of that loop body, then B' language-rediscovers this model. The only requirement on the log is that it is directly-follows complete w.r.t. the model underlying it. We argued that B' returns the smallest, most-precise, most-general model adhering to the model restrictions, and runs in a time polynomial to the number of activities and the size of the log.

Future Work. It might be possible to drop the model restriction 2 of Section 6.4, which requires that the the sets of start and end activities of the leftmost branch of a loop operator must be disjoint, when length-two-loops are taken into account and a stronger completeness requirement is put on the log. Moreover, using another strengthened completeness assumption on the log, the no-τ restriction might be unnecessary. We plan on performing an empirical study to compare our B' algorithm to existing techniques. Noise, behaviour in the log that is not in the underlying model, could be handled by filtering the directly-follows relation, in a way comparable to the Heuristics miner [24], before constructing the directly-follows graph.

References

1. van der Aalst, W.: Workflow verification: Finding control-flow errors using petri-net-based techniques. In: Business Process Management, pp. 161–183. Springer (2000)
2. van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
3. van der Aalst, W., Buijs, J., van Dongen, B.: Improving the representational bias of process mining using genetic tree mining. SIMPDA 2011 Proceedings (2011)
4. van der Aalst, W., de Medeiros, A., Weijters, A.: Genetic process mining. Applications and Theory of Petri Nets 2005 pp. 985–985 (2005)
5. van der Aalst, W., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. Knowledge and Data Engineering, IEEE Transactions on 16(9), 1128–1142 (2004)
6. Badouel, E.: On the α-reconstructibility of workflow nets. In: Petri Nets'12. LNCS, vol. 7347, pp. 128–147. Springer (2012)
7. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. Business Process Management pp. 375–383 (2007)
8. Bergenthum, R., Desel, J., Mauser, S., Lorenz, R.: Synthesis of Petri nets from term based representations of infinite partial languages. Fundam. Inform. 95(1), 187–217 (2009)
9. Buijs, J., van Dongen, B., van der Aalst, W.: A genetic algorithm for discovering process trees. In: Evolutionary Computation (CEC), 2012 IEEE Congress on. pp. 1–8. IEEE (2012)
10. Buijs, J., van Dongen, B., van der Aalst, W.: On the role of fitness, precision, generalization and simplicity in process discovery. In: On the Move to Meaningful Internet Systems: OTM 2012, pp. 305–322. Springer (2012)

11. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving Petri nets from finite transition systems. Computers, IEEE Transactions on 47(8), 859–882 (1998)
12. van Dongen, B., de Medeiros, A., Verbeek, H., Weijters, A., van der Aalst, W.: The ProM framework: A new era in process mining tool support. Applications and Theory of Petri Nets 2005 pp. 1105–1116 (2005)
13. van Dongen, B., de Medeiros, A., Wen, L.: Process mining: Overview and outlook of Petri net discovery algorithms. Transactions on Petri Nets and Other Models of Concurrency II pp. 225–242 (2009)
14. Fahland, D., van der Aalst, W.: Repairing process models to reflect reality. In: BPM'12. LNCS, vol. 7481, pp. 229–245. Springer (2012)
15. Fahland, D., Favre, C., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Analysis on demand: Instantaneous soundness checking of industrial business process models. Data Knowl. Eng. 70(5), 448–466 (2011)
16. Gambini, M., La Rosa, M., Migliorini, S., ter Hofstede, A.: Automated error correction of business process models. In: BPM'11. LNCS, vol. 6896, pp. 148–165. Springer (2011)
17. Günther, C., van der Aalst, W.: Fuzzy mining–adaptive process simplification based on multi-perspective metrics. Business Process Management pp. 328–343 (2007)
18. Leemans, S., Fahland, D., van der Aalst, W.: Discovering block-structured process models from event logs - a constructive approach. Tech. Rep. BPM-13-06, Eindhoven University of Technology (April 2013)
19. de Medeiros, A., Weijters, A., van der Aalst, W.: Genetic process mining: an experimental evaluation. Data Mining and Knowledge Discovery 14(2), 245–304 (2007)
20. Polyvyanyy, A., Garcia-Banuelos, L., Fahland, D., Weske, M.: Maximal structuring of acyclic process models. The Computer Journal (2012), http://comjnl.oxfordjournals.org/content/early/2012/09/19/comjnl.bxs126.abstract
21. Polyvyanyy, A., Vanhatalo, J., Völzer, H.: Simplified computation and generalization of the refined process structure tree. In: WS-FM'10. LNCS, vol. 6551, pp. 25–41. Springer (2010)
22. Reisig, W., Schnupp, P., Muchnick, S.: Primer in Petri Net Design. Springer-Verlag New York, Inc. (1992)
23. Rozinat, A., de Medeiros, A., Günther, C., Weijters, A., van der Aalst, W.: The need for a process mining evaluation framework in research and practice. In: Business Process Management Workshops. pp. 84–89. Springer (2008)
24. Weijters, A., van der Aalst, W., de Medeiros, A.: Process mining with the heuristics miner-algorithm. Technische Universiteit Eindhoven, Tech. Rep. WP 166 (2006)
25. Wen, L., van der Aalst, W., Wang, J., Sun, J.: Mining process models with non-free-choice constructs. Data Mining and Knowledge Discovery 15(2), 145–180 (2007)
26. Wen, L., Wang, J., Sun, J.: Mining invisible tasks from event logs. Advances in Data and Web Management pp. 358–365 (2007)
27. van der Werf, J., van Dongen, B., Hurkens, C., Serebrenik, A.: Process discovery using integer linear programming. Applications and Theory of Petri Nets pp. 368–387 (2008)