Discovering Block-Structured Process Models from Incomplete Event Logs

Sander J.J. Leemans, Dirk Fahland, and Wil M.P. van der Aalst

Eindhoven University of Technology, the Netherlands
{s.j.j.leemans, d.fahland, w.m.p.v.d.aalst}@tue.nl

Abstract One of the main challenges in process mining is to discover a process model describing observed behaviour in the best possible manner. Since event logs only contain example behaviour and one cannot assume to have seen all possible process executions, process discovery techniques need to be able to handle incompleteness. In this paper, we study the effects of such incomplete logs on process discovery. We analyse the impact of incompleteness of logs on behavioural relations, which are an abstraction often used by process discovery techniques. We introduce probabilistic behavioural relations that are less sensitive to incompleteness, and exploit these relations to provide a more robust process discovery algorithm. We prove this algorithm to be able to rediscover a model of the original system. Furthermore, we show in experiments that our approach even rediscovers models from incomplete event logs that are much smaller than required by other process discovery algorithms.

Keywords: process discovery, block-structured process models, rediscoverability, process trees

1 Introduction

Organisations nowadays collect and store considerable amounts of event data. For instance, workflow management systems log audit trails, and enterprise resource planning systems store transaction logs. From these event logs, process mining aims to extract information, such as business process models, social networks, bottlenecks and compliance with regulations [1]. In this paper we focus on the most challenging problem: discovering a process model from example traces. Learning a process model (e.g., a Petri net) from example traces in an event log, called *process discovery*, is one of the first and most challenging steps of process mining.

Two problems of logs are particularly challenging for process discovery algorithms. First, the log may contain *infrequent behaviour*, which forces algorithms to either exclude this behaviour or return complicated, unreadable models describing all behaviour [18]. Second, the log might contain insufficient information to discover a process model that represents the system well: the log might be *incomplete*. Incompleteness forces algorithms to either exclude the missing behaviour and reduce the as yet unseen behaviour the model can produce, or include the missing, unknown, behaviour and risk guessing wrong. In this paper, we focus on handling incomplete logs.

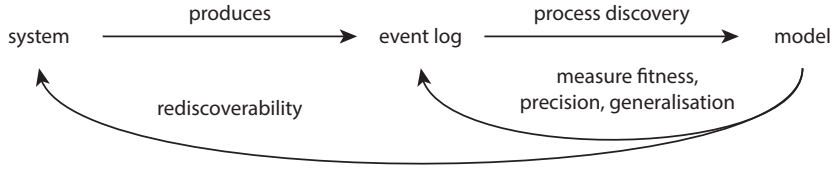


Figure 1: Traditional model quality assessment (fitness, precision, generalisation) and rediscoverability.

A notion closely related to incompleteness is rediscoverability. If a process discovery technique has *rediscoverability*, it is able to discover models that have the same language as the real-life process by which a log was produced [3,5,17]. Figure 1 shows the context of process discovery, rediscoverability, and how discovered models can be evaluated. Traditionally, models are evaluated with respect to the event log: *fitness* measures what part of the event log is described by the model, *precision* is high when the model does not allow too much behaviour that was not in the event log, and *generalisation* is high when the model allows more behaviour than just the behaviour in the event log. Although fitness, precision, and generalisation are intuitively clear, different formal definitions are possible [13,22,23]. Measuring the quality of a discovered model with respect to its event log might be useful, but whether the best model for the event log is the best model for the system is not captured by these measures. Therefore, to compare process discovery techniques it is useful to study rediscoverability, as that gives theoretical bounds to when a model is language-equivalent to its real-life system.

Rediscoverability is usually proven using assumptions about both log and model [3,5,17]. A model must be from a certain class, and a log must contain sufficient information. The notion what information is sufficient, *completeness*, depends on the discovery algorithm. Generally, the strongest completeness notion is *language-completeness*, i.e., each trace through the process must be present in the log. The weakest completeness notion is that each process step must occur at least once in the log: *activity-completeness* [17].

Typically, rediscoverability can only be guaranteed if the log is complete. In this paper, we investigate the problem of *rediscovering* process models from event logs, in particular from *incomplete* event logs.

Another desirable property of process discovery algorithms is that they return simple and sound models. A *simple* model needs few constructs to express its behaviour, and a *sound* model is a model free of deadlocks and other anomalies. While an unsound model might be useful, it is, for instance, not well suited for compliance evaluation and bottleneck analysis [18]. Therefore, in this paper we will focus on *process trees*: abstract hierarchical block-structured Petri nets that are guaranteed to be sound.

The Inductive Miner (IM) [17] is an example of an algorithm that discovers process trees and for which rediscoverability has been proven. IM applies a divide-and-conquer approach: it partitions the activities, selects the most important process construct, splits the log and recurses until a base case is encountered.

In this paper, we adapt IM to handle incomplete logs: we keep the divide-and-conquer approach, but replace the activity partition step by an optimisation problem. We intro-

duce relations between activities, estimate probabilities of these relations and search for a partition of activities that is optimal with respect to these probabilities. Rediscoverability is proven assuming log completeness and a sufficiently large log; we give a lower bound for sufficiency.

In the remainder of this paper, we first explore related work. In Section 3, we introduce logs, Petri nets, process trees and completeness notions. We study effects of incompleteness on behavioural relations in Section 4 and describe behavioural probabilisations. Section 5 describes the algorithm, Section 6 proves rediscoverability for sufficiently large logs, and illustrates how incompleteness is handled by the new approach, compared with other approaches. Section 7 concludes the paper.

2 Related Work

Petri net synthesis aims to build an equivalent Petri net from a transition system or a language. [15] introduced region theory to characterise places in a Petri net, and several synthesis methods were proposed, for instance [11,19,6,12].

Process discovery differs from Petri net synthesis in the assumption on completeness. Synthesis assumes that the complete language of the system is described in some form. For process discovery we cannot assume the log to be language-complete, as typically only a fraction of the possible behaviour can be observed in the event log, making language-completeness often impossible or infeasible. For example, the language of a model having a loop contains infinitely many traces, and the language of a model describing the parallel execution of 10 activities contains at least $10! = 3628800$ different traces [1]. In contrast, a typical log only contains a fraction of that.

Many process discovery techniques have been proposed. For instance, after a transition system has been constructed from the log, state-based region miner techniques construct a Petri net by folding regions of states into places [4,28]. Typically, state-based region techniques provide rediscoverability guarantees [10], but have problems dealing with parallelism.

Process trees, or block structures in general, have been used in process discovery, both inside the scope of Petri nets [8,2,20], as outside [24,25] the scope of Petri nets. They provide a natural, structured, well-defined way of describing processes that are often easily translatable to Petri nets. The process tree formalisms used in [8,17,18] guarantee soundness as well. Process tree discovery techniques have also been proposed before. For instance, the approach used by [26] constructs a process tree from a log by enumerating all traces, after which the process tree is simplified. The Evolutionary Tree Miner (ETM) [8] uses a genetic approach to discover a process tree, i.e., a random population is mutated until a certain stop criterion is met, but as it is steered by log-based metrics, fitness, precision, generalisation and simplicity, and by its random nature, it is unable to guarantee rediscoverability. A natural strategy when using block structures is to apply a divide-and-conquer strategy, which has been applied to process discovery in for instance [9,36,17,18].

Behavioural relations have been proven to be able to distinguish languages of classes of Petri nets [30], and they have been used to refine or coarsen models, i.e., making them more or less abstract [27,16], to compare process models [29], and to perform

process discovery. For instance, the behavioural relation used in the α algorithm [3], its derivatives [33,34], and in [17,18], the directly-follows relation, holds for two activities if one activity can consecutively follow the other activity. A notion close to the directly-follows relation is the eventually-follows relation, which holds if one activity can eventually be followed by another. This eventually-follows relation has been used in the context of process discovery [26,30,18].

To the best of our knowledge, the influence of incompleteness has not been systematically studied on neither behavioural relations nor process discovery.

3 Traces, Event Logs, Petri Nets and Completeness

Traces, Event Logs. A *trace* is a sequence of activities: $\langle a, a, b\rangle$ denotes a trace in which first a occurred, then a again and finally b. Traces can be concatenated: $\langle a, b\rangle \cdot \langle c\rangle = \langle a, b, c\rangle$. An *event log* is a multiset of traces. For instance, $[\langle a, a, b\rangle^3, \langle b, b\rangle^2]$ denotes an event log in which the trace $\langle a, a, b\rangle$ happened 3 times and $\langle b, b\rangle$ happened 2 times. The function set transforms a multiset into a set: $\mathrm{set}(L) = \{t | t \in L\}$; the function Σ gives the alphabet of the log, i.e., the activities used in it.

Petri Nets, Workflow Nets and Block-Structured Workflow Nets. A *Petri net* is a bipartite directed graph of interconnected *places* and *transitions*, in which *tokens* on places model the system state and transitions model process step execution. We use the standard semantics of Petri nets, see [21].

A *workflow net* is a Petri net having a single input and a single output place, modelling the initial and final states of the system. Moreover, each element is on a path from input to output [3]. A consecutive sequence of process executions that brings the system from the initial state into the final state, corresponds to a *trace*. The set of traces that can be produced by a model M, the *language* of M, is denoted by $\mathcal{L}(M)$.

A *block-structured workflow net* is a hierarchical workflow net: it can be divided recursively into workflow nets. An example is shown in Figure 2.

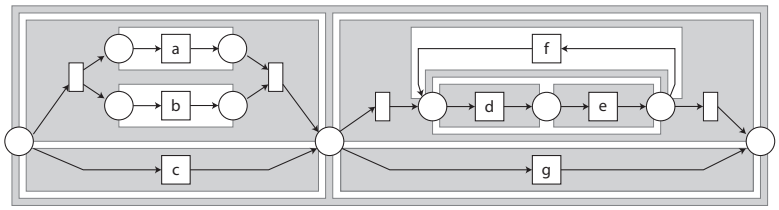


Figure 2: A block-structured workflow net M_E; filled regions denote the block-structure; process tree $\rightarrow(\times(\wedge(a, b), c), \times(\circlearrowleft(\rightarrow(d, e), f), g))$ corresponds to this net.

Process Trees. A *process tree* is an abstract hierarchical representation of a block-structured workflow net. The leaves of the tree are *activities*, representing transitions. The nodes of the tree, *operators*, describe how their children are combined. This paper uses four operators: $\times$, $\rightarrow$, $\wedge$ and $\circlearrowleft$. The $\times$ operator describes the exclusive choice between its children, $\rightarrow$ the sequential composition and $\wedge$ the parallel composition. The

first child of a $\circlearrowleft$ tree is the loop *body*, the non-first children are *redo* parts. For instance, $\circlearrowleft(a, b)$ is the composition of a trace of the body a, then zero-or-more times a trace from a redo part b and a body a again: $a((b|c)a)^*$.

Each process tree is easily translatable to a sound workflow net. For example, Figure 2 shows the block-structured workflow net corresponding to the process tree $M_E = \rightarrow(\times(\wedge(a, b), c), \times(\circlearrowleft(\rightarrow(d, e), f), g))$.

To define the semantics of process trees, we assume a finite set of activities Σ to be given. The language of an activity is the execution of that activity (a process step). The language of the *silent activity* τ contains only the empty trace: executing τ adds nothing to the log. The language of an operator is a combination of the languages of its children.

To characterise $\wedge$, we use the shuffle product $S_1 \sqcup\!\sqcup \ldots S_n$, which takes sets of traces from $S_1 \ldots S_n$ and interleaves their traces $t_1 \in S_1, \ldots, t_n \in S_n$ while maintaining the partial order within each t_i [7]. For instance,

$$\{\langle a, b\rangle\} \sqcup\!\sqcup \{\langle c, d\rangle\} = \{\langle a, b, c, d\rangle, \langle a, c, b, d\rangle, \langle a, c, d, b\rangle, \\ \langle c, d, a, b\rangle, \langle c, a, d, b\rangle, \langle c, a, b, d\rangle\}$$

Using this notation, we define the semantics of process trees:

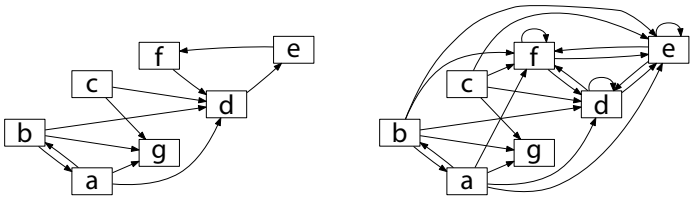
$$\begin{aligned}
\mathcal{L}(\tau) &= \{\langle\,\rangle\} \\
\mathcal{L}(a) &= \{\langle a\rangle\} \text{ for } a \in \Sigma \\
\mathcal{L}(\times(M_1, \ldots, M_n)) &= \mathcal{L}(M_1)|\mathcal{L}(M_2)\ldots\mathcal{L}(M_n) \\
\mathcal{L}(\rightarrow(M_1, \ldots, M_n)) &= \mathcal{L}(M_1) \cdot \mathcal{L}(M_2) \cdot \cdots \mathcal{L}(M_n) \\
\mathcal{L}(\wedge(M_1, \ldots, M_n)) &= \mathcal{L}(M_1) \sqcup\!\sqcup \mathcal{L}(M_2)\ldots\mathcal{L}(M_n) \\
\mathcal{L}(\circlearrowleft(M_1, \ldots, M_n)) &= \mathcal{L}(M_1)(\mathcal{L}(\times(M_2, \ldots, \mathcal{L}(M_n))\mathcal{L}(M_1))^*
\end{aligned}$$

As an example, the language of M_E is $(ab|ba|c)(de(fde)^*|g)$. The function Σ gives the alphabet of a process tree: $\Sigma(M_E) = \{a, b, c, d, e, f, g\}$. We use $\bigoplus$ to denote the set of operators, and often $\oplus$ to denote a process tree operator: $\oplus \in \bigoplus$, $\bigoplus = \{\times, \rightarrow, \wedge, \circlearrowleft\}$. Obviously, the order of children for $\times$ and $\wedge$ and the order of non-first children of $\circlearrowleft$ is arbitrary.

Directly-Follows Relation, Transitive Closure and Graphs. The *directly-follows relation* $\mapsto$ has been proposed in [3] as an abstraction of the behaviour described by a model or a log. From a model M, take two activities a and b. If b can follow a directly in M, $\langle \ldots, a, b, \ldots\rangle \in \mathcal{L}(M)$, then $a \mapsto_M b$. For a log L, $\mapsto_L$ is defined similarly. For logs, $\mapsto$ is monotonic: for a pair of activities, $\mapsto$ cannot cease to hold by adding more traces to the log.

A $\mapsto$-*path* is a sequence $a_1 \ldots a_k$ of activities such that $k \geqslant 2$ and $\forall_{1 \leqslant i < k} a_i \mapsto a_{i+1}$. The transitive closure of $\mapsto$ is denoted by $\mapsto^+$: for activities a and b, the relation $a \mapsto^+ b$ holds if there exists a $\mapsto$-path from a to b. [1] For a model M (resp. a log L), $Start(M)$

[1] We did not choose the eventually-follows/weak-order relation [18,30], as its completeness does not survive log splitting; Lemma 11 does not hold for it.



(a) $\mapsto$-graph of M_E (b) $\mapsto^+$-graph of M_E

Figure 3: Graphs of M_E showing its directly-follows relation $\mapsto$ and its transitive closure $\mapsto^+$.

(resp. $Start(L)$) denotes the start activities, found at the beginning of a trace, and $End(M)$ (resp. $End(L)$) the end activities, that can conclude a trace.

Figure 3a shows the directly-follows relation of M_E in graph notation: a directly-follows graph. In this graph, an edge is drawn between a pair of activities (x, y) if $x \mapsto y$. Similarly, Figure 3b shows the graph of $\mapsto^+$ for M_E.

Completeness. Using these relations, we introduce two completeness notions, between a model M and a log L:

- L is *activity complete* to M ($L \diamond_\Sigma M$), if each activity of M is present in L at least once: $\Sigma(M) \subseteq \Sigma(L)$.
- L is *directly-follows complete* to M ($L \diamond_\mapsto M$), if L is activity-complete to M, its directly-follows relation is complete, and both start and end activities are complete: $L \diamond_\Sigma M$, $\mapsto_M \subseteq \mapsto_L$, $Start(M) \subseteq Start(L)$ and $End(M) \subseteq End(L)$.

Partitions and Cuts. A *partition* is a distribution of an activity set Σ into disjoint non-empty subsets $\Sigma_1 \dots \Sigma_n$, with $n > 1$. A pair of activities (a, b) is *partitioned* by a partition $\Sigma_1, \dots, \Sigma_n$ if a and b are not both in the same Σ_i. A *cut* is a partition together with a process tree operator, for example $(\rightarrow, \{a\}, \{b, c, d, e, f\})$. If a pair of activities is partitioned by the partition in a cut, the pair *crosses* the cut.

Obviously, any process tree can be rewritten to a language-equivalent binary process tree. Therefore, without loss of generality, in this paper we consider only binary partitions and cuts.

4 Behavioural Relations

In many Petri net discovery algorithms, such as [3,17,18,33,34], a two-stage approach is used: first, an abstraction of the log is derived, and second, from this abstraction a model is generated. The directly-follows relation $\mapsto$ is often used as a behavioural relation. In this section, we first describe the influence of incompleteness on behavioural relations. In order to do that, we classify pairs of activities inspired by the process tree operators, by using the $\mapsto$ relation, after which we show the effect incompleteness has on this classification. Second, we introduce a probabilistic version of the classification that helps discovery techniques deal with incompleteness.

Figure 4 identifies nine cases for $\mapsto$ and $\mapsto^+$ between two given activities a and b, and organises these cases in a lattice. The structure of the lattice follows from $\mapsto$ and $\mapsto^+$: an edge in the lattice corresponds to an extension of the $\mapsto$-relation with one pair of activities.

The lattice yields five relations between activities: the commutative $\underline{\times}$, $\underline{\wedge}$ and $\underline{\circlearrowleft}_{\mathrm{i}}$, and the non-commutative $\underline{\rightarrow}$ and $\underline{\circlearrowleft}_{\mathrm{s}}$. For instance, if $b \mapsto a$ and $a \not\mapsto^+ b$, then $\underline{\rightarrow}(a, b)$, and if $a \mapsto^+ b$, $b \mapsto^+ a$, $a \not\mapsto b$ and $b \not\mapsto a$, then $\underline{\circlearrowleft}_{\mathrm{i}}(a, b)$. Informally, $\underline{\times}(a, b)$ denotes that a and b are in an exclusive choice relation, $\underline{\rightarrow}(a, b)$ denotes that a and b are in a sequence relation, and $\underline{\wedge}(a, b)$ denotes that a and b are in a parallel relation. These are similar to the α-relations $\#_W$, $\rightarrow_W$ and $\|_W$ [3], but act globally instead of locally.

Both $\underline{\circlearrowleft}_{\mathrm{i}}(a, b)$ and $\underline{\circlearrowleft}_{\mathrm{s}}(a, b)$ denote that a and b are in a loop relation. If we would combine them into a single relation, this single relation would not give sufficient information to partition the activities. Using the two relations $\underline{\circlearrowleft}_{\mathrm{s}}$ and $\underline{\circlearrowleft}_{\mathrm{i}}$ does, as will be proven in Section 6.

We consider the commutative cases, for instance $\underline{\wedge}(a, b)$ and $\underline{\wedge}(b, a)$, to be equivalent.

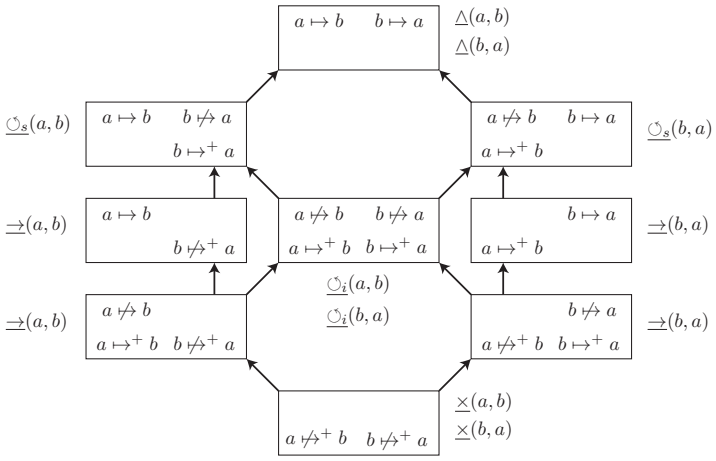
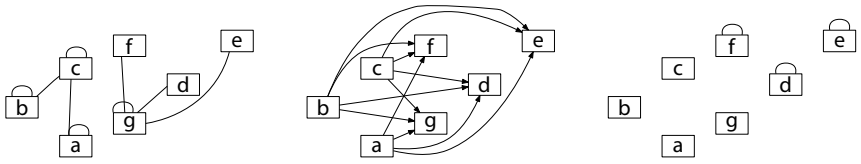


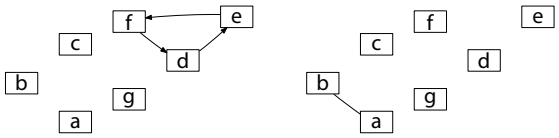
Figure 4: Activity relations; the arrows define a lattice. Implied $\not\mapsto$ and $\mapsto^+$ are omitted for readability reasons.

Consider again Petri net M_E shown in Figure 2. Figure 5 shows the activity relations of M_E as graphs. Consider the log $L_E = [\langle c, d, e, f, d, e, f, d, e\rangle, \langle b, a, d, e\rangle, \langle a, b, d, e, f, d, e\rangle, \langle c, g\rangle]$, which we produced using M_E, but L_E is not directly-follows complete to M_E, as $a \mapsto g$, $b \mapsto g$, $a \mapsto^+ g$ and $b \mapsto^+ g$ hold in M_E but not in L_E. Therefore, $\underline{\times}(a, g)$ and $\underline{\times}(b, g)$ hold in L_E; Figure 6 shows how $\underline{\times}$ and $\underline{\rightarrow}$ change. For L_E, a process discovery algorithm will regard a and b to be exclusive to g, while M_E puts them in sequence, and thus be unable to rediscover M_E. The problem illustrated with these activity relations is inherent to any process discovery algorithm using behavioural relations; any technique that just uses behavioural relations is likely unable to rediscover M_E from the directly-follows incomplete L_E.

In the following, we explore ways to use information from incomplete logs that could help to rediscover the original model. Therefore, in the remainder of this paper we as-

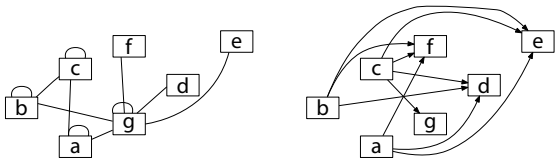


(a) $\underline{\times}$-graph (b) $\underline{\rightarrow}$-graph (c) $\underline{\circlearrowleft_{\mathrm{i}}}$-graph



(d) $\underline{\circlearrowleft_{\mathrm{s}}}$-graph (e) $\underline{\wedge}$-graph

Figure 5: Activity relations of M_E as graphs. In the $\underline{\rightarrow}$-graph a directed edge is drawn from a to b if $\underline{\rightarrow}(a, b)$ holds, and similar for $\underline{\circlearrowleft_{\mathrm{s}}}$. For $\underline{\times}$, $\underline{\wedge}$ and $\underline{\circlearrowleft_{\mathrm{i}}}$, which are commutative, undirected edges are drawn.



(a) $\underline{\times}$-graph (b) $\underline{\rightarrow}$-graph

Figure 6: Two activity relations of L_E as graphs. Notice that $\rightarrow(a, g)$ and $\rightarrow(b, g)$ do not hold anymore, while $\times(a, g)$ and $\times(b, g)$ now do.

sume that the log only contains behaviour from its system, i.e., no noise is present. First, some information in the log may allow us to conclude that a particular relation between two activities cannot hold. For instance, if the log contains a trace $\langle b, a \rangle$, then $\underline{\rightarrow}(a, b)$ cannot hold. These violations follow from Figure 4: if the log contains information that a relation $\underline{\oplus}$ holds, then any weaker relation, i.e., not reachable from $\underline{\oplus}$, cannot hold; one can only move up in the lattice.

Second, the idea is to rather use an estimated probability that a relation holds than a binary choice, an idea also used in for instance the Heuristics miner [31,32]. For each of the activity relations $\underline{\oplus}$, we introduce a probabilistic version $\mathrm{p}_{\underline{\oplus}}$: for activities a and b, $\mathrm{p}_{\underline{\oplus}}(a, b)$ denotes an artificially estimated probability that (a, b) are in a $\underline{\oplus}$-related. Using the probabilistic versions makes it easier for techniques to handle incompleteness: in our example, instead of a binary choice whether $\underline{\rightarrow}(a, g)$ and $\underline{\rightarrow}(b, g)$ hold or not, we can compare the probabilities $\mathrm{p}_{\underline{\rightarrow}}$ and $\mathrm{p}_{\underline{\times}}$ to make a choice.

Our choice for these $\mathrm{p}_{\underline{\oplus}}$ is shown in Table 1. Let M be a model and L a log of M. Then, using Figure 4, we distinguish three cases and choose $\mathrm{p}_{\underline{\oplus}}(a, b)$ as follows:

- if $\underline{\oplus}(a,b)$ holds in L, it makes sense to choose $\mathrm{p}_{\underline{\oplus}}(a,b)$ as the highest of all relations for the pair (a,b). The more frequent activities a and b occur in L, the more confident we are that $\underline{\oplus}(a,b)$ holds for M, and not some stronger relation. We choose $\mathrm{p}_{\underline{\oplus}}(a,b)$ as follows: let $z(a,b) = \frac{|a|+|b|}{2}$ denote the average number of occurrences of a and b, then we define $\mathrm{p}_{\underline{\oplus}}(a,b) = 1 - \frac{1}{z(a,b)+1}$, yielding a number between $\frac{1}{2}$ and 1.
- if some relation $\underline{\otimes}(a,b)$, holds in L from which $\underline{\oplus}(a,b)$ is unreachable, then L contains a violation to $\mathrm{p}_{\underline{\oplus}}(a,b)$, so $\mathrm{p}_{\underline{\oplus}}(a,b)$ should be low. We choose $\mathrm{p}_{\underline{\oplus}}(a,b) = 0$.
- if some relation $\underline{\otimes}'(a,b)$ holds in L from which $\underline{\oplus}(a,b)$ can be reached, i.e., $\mathrm{p}_{\underline{\oplus}}(a,b)$ could hold by adding more traces to L, we choose to divide the remaining $\frac{1}{z(a,b)+1}$ evenly over all remaining entries, such that the probabilities for each pair (a,b) sum up to 1.

For example, in case of L_E, we get $\mathrm{p}_{\underline{\times}}(a,g) = 0.6$ and $\mathrm{p}_{\underline{\rightarrow}}(a,g) = 0.07$.

Table 1: Our proposal for probabilistic activity relations for activities a and b, with $z = (|a| + |b|)/2$. Negations of relations are omitted from the first column.

	$\mathrm{p}_{\underline{\times}}(a,b)$	$\mathrm{p}_{\underline{\rightarrow}}(a,b)$	$\mathrm{p}_{\underline{\rightarrow}}(b,a)$	$\mathrm{p}_{\underline{\circlearrowleft_{\mathrm{i}}}}(a,b)$	$\mathrm{p}_{\underline{\circlearrowleft_{\mathrm{s}}}}(a,b)$	$\mathrm{p}_{\underline{\circlearrowleft_{\mathrm{s}}}}(b,a)$	$\mathrm{p}_{\underline{\wedge}}(a,b)$
(nothing)	$1 - \frac{1}{z+1}$	$\frac{1}{6} \cdot \frac{1}{z+1}$	$\frac{1}{6} \cdot \frac{1}{z+1}$	$\frac{1}{6} \cdot \frac{1}{z+1}$	$\frac{1}{6} \cdot \frac{1}{z+1}$	$\frac{1}{6} \cdot \frac{1}{z+1}$	$\frac{1}{6} \cdot \frac{1}{z+1}$
$a \mapsto^+ b$	0	$1 - \frac{1}{z+1}$	0	$\frac{1}{4} \cdot \frac{1}{z+1}$	$\frac{1}{4} \cdot \frac{1}{z+1}$	$\frac{1}{4} \cdot \frac{1}{z+1}$	$\frac{1}{4} \cdot \frac{1}{z+1}$
$b \mapsto^+ a$	0	0	$1 - \frac{1}{z+1}$	$\frac{1}{4} \cdot \frac{1}{z+1}$	$\frac{1}{4} \cdot \frac{1}{z+1}$	$\frac{1}{4} \cdot \frac{1}{z+1}$	$\frac{1}{4} \cdot \frac{1}{z+1}$
$a \mapsto^+ b \wedge b \mapsto^+ a$	0	0	0	$1 - \frac{1}{z+1}$	$\frac{1}{3} \cdot \frac{1}{z+1}$	$\frac{1}{3} \cdot \frac{1}{z+1}$	$\frac{1}{3} \cdot \frac{1}{z+1}$
$a \mapsto b$	0	$1 - \frac{1}{z+1}$	0	0	$\frac{1}{2} \cdot \frac{1}{z+1}$	0	$\frac{1}{2} \cdot \frac{1}{z+1}$
$a \mapsto b \wedge b \mapsto^+ a$	0	0	0	0	$1 - \frac{1}{z+1}$	0	$\frac{1}{z+1}$
$b \mapsto a$	0	0	$1 - \frac{1}{z+1}$	0	0	$\frac{1}{2} \cdot \frac{1}{z+1}$	$\frac{1}{2} \cdot \frac{1}{z+1}$
$b \mapsto a \wedge a \mapsto^+ b$	0	0	0	0	0	$1 - \frac{1}{z+1}$	$\frac{1}{z+1}$
$a \mapsto b \wedge b \mapsto a$	0	0	0	0	0	0	1

In the next section, we demonstrate how to use any system of probabilistic relations in a concrete algorithm; one could define Table 1 differently, as long as for each pair of activities (a,b) and each relation $\underline{\oplus}$, a probability $\mathrm{p}_{\underline{\oplus}}(a,b)$ is available. In Section 6, we will show that our choices for $\mathrm{p}_{\underline{\oplus}}$ lead to a correct algorithm. We expect that the proofs given in Section 6 to extend easily to other choices, but the precise class of acceptable $\mathrm{p}_{\underline{\oplus}}$ needs further research.

5 Algorithm

In this section, we demonstrate how the probabilistic activity relations defined in Section 4 can be used to discover process trees.

We use a divide-and-conquer approach and adapt ideas from IM [17] to introduce a new disovery algorithm that we call Inductive Miner - incompleteness (IMin). IMin consists of three steps that are applied recursively: first, the $\mapsto$-graph of the log and its transitive closure $\mapsto^+$ are computed. Second, a cut is chosen such that the relations between pairs crossing the cut have the highest probability according to Table 1. The operator of the chosen cut is recorded. Third, using the cut, the log is split into a sublog

for each part and on each sublog, IMin recurses. The recursion ends when a base case, a log containing just a single activity, is encountered. The hierarchy of recorded operators is a process tree.

We first describe how to accumulate the probabilities of Table 1 to assess the probability of a cut. Second, we give the algorithm, an example and a description of our implementation.

5.1 Accumulated Estimated Probabilities for Cuts

Given activity relation probabilities, such as the ones defined in Table 1, we compute an accumulated probability for a cut. Informally, for $\oplus \in \{\times, \rightarrow, \wedge\}$, the accumulated probability $\mathrm{p}_{\oplus}$ is the average $\mathrm{p}_{\underline{\oplus}}$ over all partitioned pairs of activities.

Definition 1 (accumulated probability for $\times$, $\rightarrow$ and $\wedge$). *Let $c = (\oplus, \Sigma_1, \Sigma_2)$ be a cut, with $\oplus \in \{\times, \rightarrow, \wedge\}$. Then $\mathrm{p}_{\oplus}(\Sigma_1, \Sigma_2)$ denotes the accumulated probability of c:*

$$\mathrm{p}_{\oplus}(\Sigma_1, \Sigma_2) = \frac{\sum_{a \in \Sigma_1, b \in \Sigma_2} \mathrm{p}_{\underline{\oplus}}(a, b)}{|\Sigma_1| \cdot |\Sigma_2|}$$

Note that a $\rightarrow$, $\times$, or $\wedge$ cut requires all pairs of activities to be in the same relation sufficiently often. For a loop cut, this is not sufficient, as all crossing pairs of activities in a loop are in a loop relation ($\underline{\circlearrowleft_{\mathrm{s}}} \cup \underline{\circlearrowleft_{\mathrm{i}}}$). This loop relation suffices to describe the probability whether all activities are indeed in a loop, but on its own cannot distinguish the body of a loop from its redo parts. For this, we have to explicitly pick the start and end activities of the redo parts, such that a *redo start activity* follows a *body end activity*, and a redo end activity is followed by a body start activity. This direct succession in a loop is expressed in $\underline{\circlearrowleft_{\mathrm{s}}}$. Hence, we obtain the following probability that $c = (\circlearrowleft, \Sigma_1, \Sigma_2)$ is a loop cut for the chosen redo start activities S_2 and loop redo end activities E_2; the start and end activities of the body are the start and end activities of the log. In the next section, we show how S_2 and E_2 could be chosen.

Definition 2 (accumulated probability for $\circlearrowleft$). *Let $c = (\circlearrowleft, \Sigma_1, \Sigma_2)$ be a cut, L be a log, and S_2, E_2 be sets of activities. We aggregate over three parts: start of a redo part, end of a redo part and everything else:*

$$\begin{aligned}
redo_{start} &= \sum_{(a,b) \in End(L) \times S_2} \mathrm{p}_{\underline{\circlearrowleft_{\mathrm{s}}}}(a, b) \\
redo_{end} &= \sum_{(a,b) \in E_2 \times Start(L)} \mathrm{p}_{\underline{\circlearrowleft_{\mathrm{s}}}}(a, b) \\
indirect &= \sum_{\substack{a \in \Sigma_1, b \in \Sigma_2 \\ (a,b) \notin (End(L) \times S_2) \cup (E_2 \times Start(L))}} \mathrm{p}_{\underline{\circlearrowleft_{\mathrm{i}}}}(a, b)
\end{aligned}$$

Then, $\mathrm{p}_{\circlearrowleft}(\Sigma_1, \Sigma_2, S_2, E_2)$ denotes the accumulated probability of c:

$$\mathrm{p}_{\circlearrowleft}(\Sigma_1, \Sigma_2, S_2, E_2) = \frac{redo_{start} + redo_{end} + indirect}{|\Sigma_1| \cdot |\Sigma_2|}$$

In this definition, $redo_{start}$ and $redo_{end}$ capture the strength of S_2 and E_2 really being the start and end of the redo parts; $indirect$ captures the strength that all other pairs of activities that cross Σ_1, Σ_2 are in a loop relation.

For readability reasons, in the following, we will omit the parameters S_2 and E_2.

5.2 The Algorithm: Inductive Miner - incompleteness (IMin)

Next, we introduce a process discovery algorithm that uses the accumulated estimations of definitions 1 and 2 in a divide-and-conquer approach.

For this, we introduce a parameter that influences a threshold of acceptable incompleteness. By default, a cut with highest $\mathrm{p}_\oplus$ is to be selected at all times. However, a low $\mathrm{p}_\oplus$ might indicate that the behaviour in the log cannot be described well by a block-structured Petri net. Therefore, a parameter h is included: if there is no cut with $\mathrm{p}_\oplus \geqslant h$, a flower model $\circlearrowleft(\tau, a_1, \ldots, a_m)$ with $\{a_1, \ldots, a_m\} = \Sigma(L)$, allowing for any trace over $\Sigma(L)$ [17], is returned.

function IMIN(L)
  if $L = [\langle a \rangle^x]$ with $a \in \Sigma$ and $x \geqslant 1$ **then**
    return a
  end if
  $(\oplus, \Sigma_1, \Sigma_2) \leftarrow$ cut of $\Sigma(L)$ with highest $\mathrm{p}_\oplus(\Sigma_1, \Sigma_2)$; $\oplus \in \bigoplus$
  if $\mathrm{p}_\oplus(\Sigma_1, \Sigma_2) \geqslant h$ **then**
    $L_1, L_2 \leftarrow$ SPLIT$(L, (\oplus, \Sigma_1, \Sigma_2))$
    return $\oplus$(IMin(L_1), IMin(L_2))
  else
    return $\circlearrowleft(\tau, a_1, \ldots, a_m)$ where $\{a_1, \ldots, a_m\} = \Sigma(L)$
  end if
end function

IMin contains two non-trivial operations: selecting a cut with highest $\mathrm{p}_\oplus$ and the SPLIT function. To select a cut with highest $\mathrm{p}_\oplus$, and in case of $\circlearrowleft$ to choose S_2 and E_2, our implementation uses an SMT-solver. For more details of the translation to SMT, please refer to Appendix A.

The function SPLIT splits a log L into sublogs L_1 and L_2, according to a given cut $c = (\oplus, \Sigma_1, \Sigma_2)$, by projecting the traces of L on Σ_1 and Σ_2. For example, SPLIT applied to a sequence cut $(\rightarrow, \{a\}, \{b\})$ and a trace $\langle a, a, b, b \rangle$ yields $\langle a, a \rangle$ and $\langle b, b \rangle$. In addition, for $\circlearrowleft$, traces are split on the points where the trace 'leaves' Σ_1 and 'enters' Σ_2. For example: SPLIT$([\langle a, b, a, a, b, a \rangle], (\circlearrowleft, \{a\}, \{b\}))$ yields $[\langle a \rangle^2, \langle a, a \rangle]$ and $[\langle b \rangle^2]$. For a more detailed formal description, please refer to [17].

IMin has been implemented as part of the Inductive Miner plug-in of the ProM framework [14], available at http://www.promtools.org.

Example 3. As an example, consider again the log $L_E = [\langle c, d, e, f, d, e, f, d, e \rangle, \langle b, a, d, e \rangle, \langle a, b, d, e, f, d, e \rangle, \langle c, g \rangle]$. If IMin is applied to L_E with $h = 0$, the first most likely cut is $(\rightarrow, \{a, b, c\}, \{d, e, f, g\})$, with a $\mathrm{p}_\rightarrow$ of about 0.64. The choice for $\rightarrow$ is recorded, and L_E is split into $[\langle c \rangle^2, \langle b, a \rangle, \langle a, b \rangle]$ and $[\langle d, e, f, d, e, f, d, e \rangle, \langle d, e \rangle, \langle d, e, f, d, e \rangle, \langle g \rangle]$. Then, IMin recurses on both these sublogs. Figure 7 shows the recursive steps that

are taken by IMin. The final result is $\rightarrow(\times(\wedge(a,b),c),\times(\circlearrowleft(\rightarrow(d,e),f),g))$, which is equal to M_E.

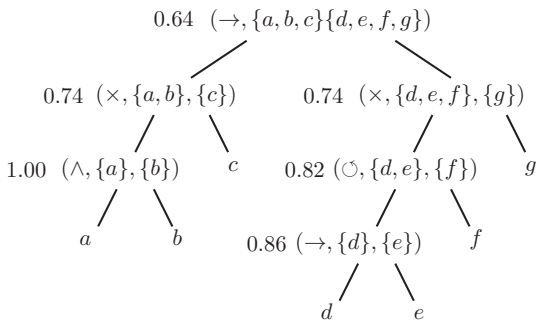


Figure 7: Running example: IMin(L_E). As a first step, the cut with highest $\mathrm{p}_\oplus$ is $(\rightarrow, \{a,b,c\}, \{d,e,f,g\})$, with $\mathrm{p}_\oplus = 0.64$. Then, IMin recurses as shown.

6 Rediscoverability

In this section, we report on the rediscoverability of IMin. We first describe a class of process trees, for which we then prove that IMin has rediscoverability, given a directly-follows complete log in which each activity occurs sufficiently often. After that, we report on experiments showing that IMin manages to rediscover these process trees, even from smaller logs than those needed by other discovery algorithms.

6.1 Class of Rediscoverable Process Trees

The class of process trees C_R for which we will prove rediscoverability is as follows:

Definition 4 (Class C_R). *Let M be a process tree. M belongs to C_R if for each (sub)tree M' at any position in M, it holds that*

- *The subtree is not a silent activity: $M' \neq \tau$*
- *If $M' = \oplus(M'_1 \ldots M'_n)$, with $\oplus$ a process tree operator, then no activity appears more than once: $\forall_{1 \leqslant i < j \leqslant n} \Sigma(M'_i) \cap \Sigma(M'_j) = \varnothing$*
- *If $M' = \circlearrowleft(M'_1 \ldots M'_n)$, then M'_1 is required to have disjoint start and end activities: $Start(M'_1) \cap End(M'_1) = \varnothing$*

6.2 Normal Form

In order to prove language-rediscoverability, we use a language-unique normal form. Each process tree can be converted into this normal form using the following language-preserving reduction rules. If no rule can be applied to a tree, the tree is in language-unique normal form [17].

Note that the order of children of $\times$ and $\wedge$, and redo children of $\circlearrowleft$ is arbitrary.

Definition 5 (Normal Form). *Let M be a process tree. Then applying the following reduction rules exhaustively on subtrees of M yields a language-unique normal form:*

$$\begin{aligned}
\oplus(M') &\to M', \textit{with} \oplus \in \bigoplus \\
\times(\cdots_1, \times(\cdots_2), \cdots_3) &\to \times(\cdots_1, \cdots_2, \cdots_3) \\
\to(\cdots_1, \to(\cdots_2), \cdots_3) &\to \to(\cdots_1, \cdots_2, \cdots_3) \\
\wedge(\cdots_1, \wedge(\cdots_2), \cdots_3) &\to \wedge(\cdots_1, \cdots_2, \cdots_3) \\
\circlearrowleft(\circlearrowleft(M, \cdots_1), \cdots_2) &\to \circlearrowleft(M, \cdots_1, \cdots_2) \\
\circlearrowleft(M, \cdots_1, \times(\cdots_2), \cdots_3) &\to \circlearrowleft(M, \cdots_1, \cdots_2, \cdots_3)
\end{aligned}$$

Using this normal form, IMin can discover the language of any tree by searching for only binary cuts. For example, if $M = \to(M_1, M_2, M_3)$, it is perfectly fine to discover either $\to(M_1, \to(M_2, M_3))$ or $\to(\to(M_1, M_2), M_3)$.

We say that a cut c *conforms* to a model M in normal form if selecting c does not disable discovery of a tree equivalent to M:

Definition 6. *Let $c = (\oplus, \Sigma_1, \Sigma_2)$ be a cut and let $M = \oplus(M_1 \dots M_n)$ be a model in normal form. Then c* conforms *to M if no $\Sigma(M_i)$ is partitioned: $\forall_i \exists_j \Sigma(M_i) \subseteq \Sigma_j$. Moreover, for non-commutative operators, order must be maintained.*

6.3 Formal Rediscoverability

The main theorem states that any model from class C_R can be rediscovered from a log whose activities occur at least a certain number of times. Let $least(L)$ denote the number of times the least occurring activity occurs in a log L.

Theorem 7. *Assume a model M that is of class C_R. Then there exists a $k \in \mathbb{N}$ such that for all logs L with $\mathrm{set}(L) \subseteq \mathcal{L}(M)$, $L \diamond\!\!\rightarrow M$ and $least(L) \geqslant k$, it holds that $\mathcal{L}(IMin(L)) = \mathcal{L}(M)$.*

We prove the theorem as follows: we first show that IMin selects the correct root operator (Lemma 9), then that IMin selects a partition corresponding to M (Lemma 10), and finally that log splitting yields correct directly-follows complete sublogs (Lemma 11), on which IMin recurses.

In these lemmas, we will use a very general property of partitions: any two partitions share at least one pair of activities that crosses both partitions.

Lemma 8. *Take two binary partitions Σ_1, Σ_2 and Σ_1', Σ_2', both of the same Σ. Then there is a pair of activities that is partitioned by both partitions.*

Proof. Towards contradiction, assume there is no pair that is partitioned by both Σ_1, Σ_2 and Σ_1', Σ_2'. Take $a_1, a_1' \in \Sigma_1$, $a_2 \in \Sigma_2$. Pairs (a_1, a_2) and (a_1', a_2) are partitioned by Σ_1, Σ_2, so by assumption they are not partitioned by Σ_1', Σ_2'. Thus, there is an $1 \leqslant i \leqslant 2$ such that $a_1, a_1', a_2 \in \Sigma_i'$. As we posed no restrictions on a_1 and a_1', for some $1 \leqslant i \leqslant 2$, $\Sigma_1 \subseteq \Sigma_i'$. By similar reasoning, $\Sigma_2 \subseteq \Sigma_i'$, so $\Sigma_1 \cup \Sigma_2 \subseteq \Sigma_i'$. Therefore, $\Sigma_i' = \Sigma$ and hence Σ_1', Σ_2' is not a partition. □

In the following lemma, we prove that for each log for which $least$ is sufficiently large, IMin selects the correct root operator.

Lemma 9. *Assume a reduced model $M = \oplus(M_1, \ldots, M_n)$. Then there exists a $k \in \mathbb{N}$ such that for all logs L with $\mathrm{set}(L) \subseteq \mathcal{L}(M)$, $L \diamond\!\!\mapsto M$ and $least(L) \geqslant k$, it holds that IMin(L) selects $\oplus$.*

Proof. IMin selects binary cuts, while M can have an arbitrary number of children. Without loss of generality, assume that $c = (\oplus, \Sigma_1, \Sigma_2)$ is a binary cut conforming to M. Let $c' = (\otimes, \Sigma_1', \Sigma_2')$ be an arbitrary cut of M, with $\otimes \neq \oplus$. We need to prove that $\mathrm{p}_\oplus(\Sigma_1, \Sigma_2) > \mathrm{p}_\otimes(\Sigma_1', \Sigma_2')$, which we do by computing a lower bound for $\mathrm{p}_\oplus(\Sigma_1, \Sigma_2)$ and an upper bound for $\mathrm{p}_\otimes(\Sigma_1', \Sigma_2')$ and then comparing these two bounds. Apply case distinction on whether $\oplus = \circlearrowleft$:

– Case $\oplus \neq \circlearrowleft$. We start with the lower bound for $\mathrm{p}_\oplus(\Sigma_1, \Sigma_2)$. Obviously,

$$\mathrm{p}_\oplus(\Sigma_1, \Sigma_2) = \frac{\sum_{a \in \Sigma_1, b \in \Sigma_2} \mathrm{p}_{\underline{\oplus}}(a, b)}{|\Sigma_1| \cdot |\Sigma_2|}$$

By semantics of process trees, Figure 4, $\mathrm{set}(L) \subseteq \mathcal{L}(M)$ and $L \diamond\!\!\mapsto M$, for each activity pair (a, b) that crosses c, $\underline{\oplus}(a, b)$ holds. For each such pair, we chose $\mathrm{p}_{\underline{\oplus}}(a, b) \geqslant 1 - \frac{1}{z(a,b)+1}$ (note that this would be an equality, save for $\mathrm{p}_{\underline{\wedge}}(a, b)$, which is 1). Thus,

$$\mathrm{p}_\oplus(\Sigma_1, \Sigma_2) \geqslant \frac{\sum_{a \in \Sigma_1, b \in \Sigma_2} 1 - \frac{1}{z(a,b)+1}}{|\Sigma_1| \cdot |\Sigma_2|}$$

For all a and b, $z(a, b) = \frac{|a|+|b|}{2} \geqslant \min(|a|, |b|) \geqslant least(L)$. Thus,

$$\mathrm{p}_\oplus(\Sigma_1, \Sigma_2) \geqslant 1 - \frac{1}{least(L) + 1} \tag{1}$$

Next, we prove an upper bound for $\mathrm{p}_\otimes(\Sigma_1', \Sigma_2')$. Obviously,

$$\frac{\sum_{a \in \Sigma_1', b \in \Sigma_2'} \mathrm{p}_{\underline{\otimes}}(a, b)}{|\Sigma_1'| \cdot |\Sigma_2'|} = \mathrm{p}_\otimes(\Sigma_1', \Sigma_2')$$

Let (u, v) be a pair partitioned by both Σ_1, Σ_2 and Σ_1', Σ_2'. By Lemma 8, such a pair exists. For all other $(a, b) \neq (u, v)$, it holds that $\mathrm{p}_{\underline{\otimes}}(a, b) \leqslant 1$ (abusing notation a bit by combining $\underline{\circlearrowleft_\mathrm{i}}$ and $\underline{\circlearrowleft_\mathrm{s}}$), and there are $|\Sigma_1| \cdot |\Sigma_2| - 1$ of those pairs.

$$\frac{(|\Sigma_1'| \cdot |\Sigma_2'| - 1) \cdot 1 + 1 \cdot \mathrm{p}_{\underline{\otimes}}(u, v)}{|\Sigma_1'| \cdot |\Sigma_2'|} \geqslant \mathrm{p}_\otimes(\Sigma_1', \Sigma_2')$$

As (u, v) crosses c, $\underline{\oplus}(u, v)$ holds. Then by inspection of Table 1, $\mathrm{p}_{\underline{\otimes}}(u, v) \leqslant \frac{1}{z(u,v)+1}$. Define y to be $|\Sigma_1'| \cdot |\Sigma_2'|$.

$$\frac{(y - 1) + \frac{1}{z(u,v)+1}}{y} \geqslant \mathrm{p}_\otimes(\Sigma_1', \Sigma_2')$$

From $z(a,b) = \frac{|a|+|b|}{2} \geqslant 1$ follows that $\frac{1}{z(u,v)+1} \leqslant \frac{1}{2}$. Thus,

$$\frac{(y-1)+\frac{1}{2}}{y} \geqslant \mathrm{p}_{\otimes}(\Sigma_1', \Sigma_2') \tag{2}$$

Using the two bounds (1) and (2), we need to prove that

$$1 - \frac{1}{least(L)+1} > \frac{(y-1)+\frac{1}{2}}{y} \tag{3}$$

Note that y is at most $\lfloor \Sigma(M)/2 \rfloor \cdot \lceil \Sigma(M)/2 \rceil$, which allows us to choose k such that $k > 2y-1$. By initial assumption $least(L) \geqslant k$, and therefore (3) holds. Hence, $\mathrm{p}_{\oplus}(\Sigma_1, \Sigma_2) > \mathrm{p}_{\otimes}(\Sigma_1', \Sigma_2')$.

– Case $\oplus = \circlearrowleft$. Using reasoning similar to the $\oplus \neq \circlearrowleft$ case, we derive (1). We directly reuse (2) to arrive at (3) and conclude that $\mathrm{p}_{\oplus}(\Sigma_1, \Sigma_2) > \mathrm{p}_{\otimes}(\Sigma_1', \Sigma_2')$.

Thus, $\mathrm{p}_{\oplus}(\Sigma_1, \Sigma_2) > \mathrm{p}_{\otimes}(\Sigma_1', \Sigma_2')$ holds for all $\oplus$. As IMin selects the cut with highest $\mathrm{p}_{\oplus}$, IMin selects $\oplus$. □

Next, we prove that for a log L, if $least(L)$ is sufficiently large, then IMin will select a partition conforming to M.

Lemma 10. *Assume a model $M = \oplus(M_1, \ldots, M_n)$ in normal form. Let $c = (\oplus, \Sigma_1, \Sigma_2)$ be a cut conforming to M, and let $c' = (\oplus, \Sigma_1', \Sigma_2')$ be a cut not conforming to M. Then there exists a $k \in \mathbb{N}$ such that for all logs L with $\mathrm{set}(L) \subseteq \mathcal{L}(M)$, $L \diamond\!\!\mapsto M$ and $least(L) \geqslant k$, holds that $\mathrm{p}_{\oplus}(\Sigma_1, \Sigma_2) > \mathrm{p}_{\oplus}(\Sigma_1', \Sigma_2')$.*

The proof strategy for this lemma is similar to the proof of Lemma 9: we prove that at least one "misclassified" activity pair (u, v) contributes to the average $\mathrm{p}_{\oplus}(\Sigma_1', \Sigma_2')$. The detailed proof that such a pair is included as Appendix B.

As a last lemma, we show that log splitting produces correct and directly-follows complete sublogs.

Lemma 11. *Assume a model M in normal form and a log L such that $\mathrm{set}(L) \subseteq \mathcal{L}(M)$ and $L \diamond\!\!\mapsto M$. Let $c = (\oplus, \Sigma_1, \Sigma_2)$ be a cut corresponding to M, and let L_1, L_2 be the result of* SPLIT(L, c)*. Then, there exist process trees M_1 and M_2, such that $\Sigma_1 = \Sigma(M_1)$, $\Sigma_2 = \Sigma(M_2)$, the normal form of $\oplus(M_1, M_2)$ is M, $\mathrm{set}(L_1) \subseteq \mathcal{L}(M_1)$, $L_1 \diamond\!\!\mapsto M_1$, $\mathrm{set}(L_2) \subseteq \mathcal{L}(M_2)$ and $L_2 \diamond\!\!\mapsto M_2$.*

For this lemma, we use that M can be converted into a binary tree by using the reduction rules of Definition 5 reversed. As c conforms to M, it is possible to convert M to $\oplus(M_1, M_2)$ such that $\Sigma_1 = \Sigma(M_1)$ and $\Sigma_2 = \Sigma(M_2)$. The remaining part of the proof of this lemma is similar to the proof of Lemma 13 in [17]: for each operator, it is shown that SPLIT returns sublogs L_1 and L_2 with $\mathrm{set}(L_1) \subseteq \mathcal{L}(M_1)$ and $\mathrm{set}(L_2) \subseteq \mathcal{L}(M_2)$. After that, it is proven that $L_1 \diamond\!\!\mapsto M_1$ and $L_2 \diamond\!\!\mapsto M_2$.

Using these lemmas, we can prove rediscoverability for sufficiently large logs.

Proof (of Theorem 7). We prove the theorem by induction on model sizes, being $|\Sigma(M)|$.

– Base case: $M = a$. As $\mathrm{set}(L) \subseteq \mathcal{L}(M)$, $L = [\langle a \rangle^x]$ for some $x \geqslant 1$. By code inspection, $\mathcal{L}(\text{IMin}(L)) = \mathcal{L}(M)$.

– Induction step: assume that the theorem holds for all models smaller than M. By Lemma 9 and 10, IMin selects a cut $c = (\oplus, \Sigma_1, \Sigma_2)$ conforming to M. Next SPLIT(L, c) returns an L_1 and L_2. By Lemma 11, there exists process trees M_1, M_2 such that $\mathcal{L}(\oplus(M_1, M_2)) = \mathcal{L}(M)$. By Lemma 11, $\mathrm{set}(L_1) \subseteq \mathcal{L}(M_1)$, $L_1 \diamond\!\!\mapsto M_1$, $\mathrm{set}(L_2) \subseteq \mathcal{L}(M_2)$ and $L_2 \diamond\!\!\mapsto M_2$. As of the induction hypothesis and the fact that L_1 and L_2 are sufficiently large by construction, $\mathcal{L}(\oplus(\text{IMin}(L_1), \text{IMin}(L_2))) = \mathcal{L}(\oplus(M_1, M_2)) = \mathcal{L}(M)$. Because $\text{IMin}(L) = \oplus(\text{IMin}(L_1), \text{IMin}(L_2))$, there exists a $k \in \mathbb{N}$ such that if $least(L) \geqslant k$, then $\mathcal{L}(\text{IMin}(L)) = \mathcal{L}(M)$. □

In the proofs of Lemmas 9 and 10, we chose $k > 2 \cdot \lfloor \Sigma(M)/2 \rfloor \cdot \lceil \Sigma(M)/2 \rceil - 1$. This gives an upper bound for the minimal $least(L)$ required:

Corollary 12. *An upper bound for k and $least(L)$ as used in Theorem 7 is determined by the size of the alphabet Σ: $k > 2 \cdot \lfloor \Sigma(M)/2 \rfloor \cdot \lceil \Sigma(M)/2 \rceil - 1$.*

Last, the unsolved question remaining is whether directly-follows completeness of a log implies that the log is sufficiently large, and that a generalised version of Theorem 7 holds:

Conjecture 13. *Assume a model M and a log L such that $\mathrm{set}(L) \subseteq \mathcal{L}(M)$ and $L \diamond\!\!\mapsto M$. Then $\mathcal{L}(\mathit{IMin}(L)) = \mathcal{L}(M)$.*

The experimental results reported in the remainder of this paper support this conjecture.

6.4 Experimental Result

In this section, we show that IMin can rediscover models from small logs. In addition, we investigate how various process discovery algorithms, including IMin, handle incompleteness.

Experiment. In the experiment, we aim to answer three questions: 1) Can IMin rediscover the language of models? 2) How does IMin handle incomplete logs? 3) How do other algorithms handle incomplete logs?

To answer questions 1 and 2 we investigated how large the log of a given model M has to be to rediscover the language of M, by generating logs of various sizes and trying to rediscover M from these logs. For question 3, we investigated how large logs need to be for other algorithms, such that adding more traces to the log would not change the result of the algorithm.

Setup. For answering questions 1 and 2, we generated 25 random process trees with 15 activities from class C_R. For each tree M, 20 random, sufficiently large, directly-follows complete logs were generated. For each log L, we verified that $\mathcal{L}(M)$ was rediscovered from it: $\mathcal{L}(\text{IMin}(L)) = \mathcal{L}(M)$. Then we performed a binary search on L to find the smallest sublog of L from which, in normal form, M was rediscovered. These sublogs were obtained by removing traces from L, and on each smallest sublog found, we measured the number of traces and completeness of $\mapsto$.

To answer question 3, comparing IMin to other algorithms, we used a similar procedure: for each discovery algorithm D, we used the same randomly generated process

trees to find, for each tree, the smallest logs L_D such that adding more traces to L_D would always return a model $D' = D(L_D)$ (up to isomorphism). We call the model $D(L_D)$ for such a smallest log L_D a *top model* M_T. For this experiment, we considered the following discovery algorithms: Inductive Miner (IM) [17], Integer Linear Programming miner (ILP) [35], α-algorithm (α) [3], Region miner (RM) [28,4] and flower model, all plug-ins of the ProM framework [14]. The flower model was included as a baseline, as it will reach its top model if $L \diamond_\Sigma M$: it only depends on the presence of activities in the log. All miners were applied using their default settings, and for IMin h was set to 0. For both procedures, we experimentally observed that event logs with 16000 traces were directly-follows complete and sufficiently large to rediscover the original model (in case of IMin) or to find the top model (for other algorithms).

Results. Table 2 shows the results. For example, IM on average required 97% of the $\mapsto$-pairs of the model to be present in the log to discover its top model M_T. For some models, the ILP implementation we used did not return an answer. Averages are given without these models and are marked with a preceding *.

Table 2: Results of the experiments. Column 2: for how many models M was its language rediscovered in M_T, averaged over logs. Column 3: average number of traces in the smallest sublogs. Column 4: average ratio of $\mapsto$-pairs present in smallest sublogs compared to the models M.

miner	$\mathcal{L}(M) = \mathcal{L}(M_T)$	number of traces	$\mapsto$-completeness
α	0%	133.132	1.000
ILP	12%	*258.529	*0.980
RM	4%	132.896	1.000
IM	100%	85.256	0.971
IMin	100%	32.568	0.875
Flower	0%	11.620	0.641

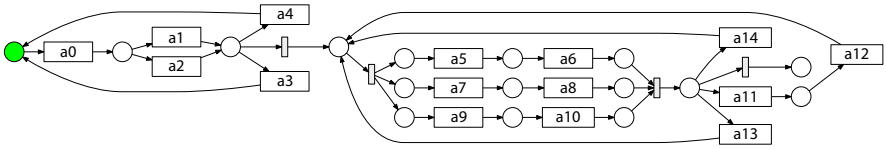
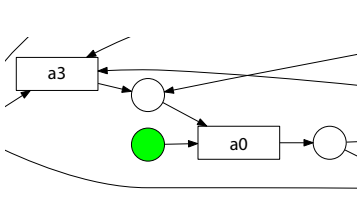


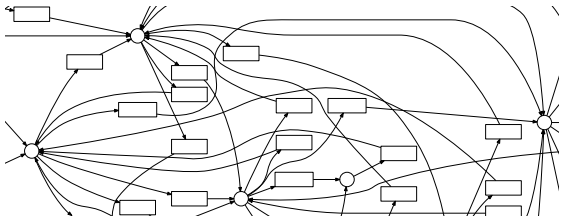
Figure 8: Petri net representation of M_F: $\rightarrow(\circlearrowleft(\rightarrow(a_0, \times(a_1, a_2)), a_3, a_4), \circlearrowleft(\wedge(\rightarrow(a_5, a_6), \rightarrow(a_7, a_8), \rightarrow(a_9, a_{10})), \rightarrow(a_{11}, a_{12}), a_{13}, a_{14})$

One of the randomly generated models is shown in Figure 8. To illustrate handling of incompleteness, we used this model to find the smallest sublog for which IMin rediscovered M_F, and applied other discovery algorithms to that sublog. The results are shown in Figure 9.

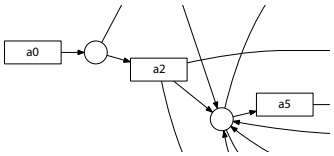
Discussion. Answering question 1, for all models and logs, IMin discovered the original model or a language-equivalent one, and even did not require the log to be directly-



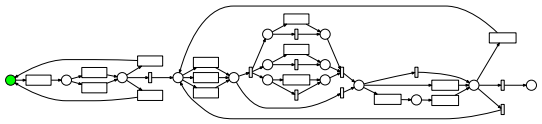
(a) Excerpt of α; a_0 cannot fire; unsound.



(b) Excerpt of RM; labels have been removed; lots of places necessary to represent parallelism.



(c) Excerpt of ILP; a_0 can fire at any time.



(d) IM; labels have been removed; misses the central parallelism.

Figure 9: Models resulting from discovery of a smallest sublog of IMin.

follows complete, which supports Conjecture 13. IMin required on average 87.5% of the $\mapsto$-relation pairs to be present in the log to discover its top model. This suggests that IMin is able to handle directly-follows incomplete logs, answering question 2.

The flower model provides a baseline: it discovers a model based on the activities that are present in a log; no process discovery technique can be expected to reach its top model without all activities being present in the log. For all models, IMin required fewer or equally many traces than any other discovery algorithm, except for the flower model, to reach its top model.

Remarkably, also IM did not require the $\mapsto$ relation to be complete at all times. A possible explanation is that log splitting might help at times. For instance, $\wedge(a, b, c)$ could be rediscovered as $\wedge(a, \wedge(b, c))$. If a log lacks $\mapsto(b, c)$, it could be introduced during log splitting: by splitting $\langle b, a, c\rangle$ with $\{a\}$ and $\{b, c\}$ yields the trace $\langle b, c\rangle$ for which $b \mapsto c$ holds, enabling the rediscovery of $\wedge(b, c)$.

Figure 9 illustrates how other discovery algorithms handle models within the representational bias of IM and IMin, for which IMin rediscovers its language. It would be interesting to see how these algorithms perform on process trees not from class C_R and on general Petri nets.

7 Conclusion

In this paper, we studied the effects of incompleteness on process discovery. We analysed the impact of incompleteness of logs on behavioural relations. We introduced probabilistic behavioural relations to make them more stable when dealing with incompleteness, and defined an algorithm based on these probabilistic relations. This algorithm was proven to be able to rediscover the language of models, given sufficiently large directly-follows complete logs. Moreover, it was shown in experiments to be able to rediscover the language of models, even when given small incomplete logs, and to need less information in the log to converge than other process discovery algorithms.

An open question remaining is whether rediscoverability holds for IMin (Conjecture 13). Another question is if directly-follows completeness is an upper bound for rediscoverability and if activity-completeness is a lower bound for it, whether these bounds are tight. The experiments we conducted suggest that there is a tighter upper bound than directly-follows completeness.

References

1. van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
2. van der Aalst, W., Buijs, J., van Dongen, B.: Improving the representational bias of process mining using genetic tree mining. SIMPDA 2011 Proceedings (2011)
3. van der Aalst, W., Weijters, A., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Trans. Knowl. Data Eng. 16(9), 1128–1142 (2004)
4. Badouel, E., Darondeau, P.: Theory of Regions. In: Lectures on Petri Nets I: Basic Models. vol. 1491, pp. 529–586 (1998)
5. Badouel, E.: On the α-reconstructibility of workflow nets. In: Petri Nets'12. LNCS, vol. 7347, pp. 128–147. Springer (2012)
6. 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)
7. Bloom, S.L., Ésik, Z.: Free shuffle algebras in language varieties. Theor. Comput. Sci. 163(1&2), 55–98 (1996)
8. 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)
9. Carmona, J.: Projection approaches to process mining using region-based techniques. Data Mining and Knowledge Discovery 24(1), 218–246 (2012)
10. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving Petri nets for finite transition systems. IEEE Trans. Computers 47(8), 859–882 (1998)
11. Darondeau, P.: Region based synthesis of p/t-nets and its potential applications. In: ICATPN. pp. 16–23 (2000)
12. Darondeau, P.: Unbounded Petri net synthesis. In: Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 413–438. Springer (2003)
13. De Weerdt, J., De Backer, M., Vanthienen, J., Baesens, B.: A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Information Systems 37, 654–676 (2012)
14. 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. Petri Nets 2005 3536, 444454 (2005)
15. Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-structures. Acta Informatica 27(4), 343–368 (1990)
16. Günther, C., van der Aalst, W.: Fuzzy mining–adaptive process simplification based on multi-perspective metrics. Business Process Management pp. 328–343 (2007)
17. Leemans, S., Fahland, D., van der Aalst, W.: Discovering block-structured process models from event logs - a constructive approach. In: Petri Nets 2013. LNCS, vol. 7927, pp. 311–329. Springer (2013)
18. Leemans, S., Fahland, D., van der Aalst, W.: Discovering block-structured process models from event logs containing infrequent behaviour. In: Business Process Management Workshops. Springer (2013), to appear
19. Lorenz, R., Mauser, S., Juhás, G.: How to synthesize nets from languages: a survey. In: Winter Simulation Conference. pp. 637–647. WSC (2007)

20. 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)
21. Reisig, W., Schnupp, P., Muchnick, S.: Primer in Petri Net Design. Springer (1992)
22. 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)
23. Rozinat, A., Veloso, M., van der Aalst, W.: Evaluating the quality of discovered process models. In: 2nd Int. Workshop on the Induction of Process Models. pp. 45–52 (2008)
24. Schimm, G.: Generic linear business process modeling. In: ER (Workshops). LNCS, vol. 1921, pp. 31–39. Springer (2000)
25. Schimm, G.: Process miner - a tool for mining process schemes from event-based data. In: JELIA. LNCS, vol. 2424, pp. 525–528. Springer (2002)
26. Schimm, G.: Mining most specific workflow models from event-based data. In: Business Process Management. LNCS, vol. 2678, pp. 25–40. Springer (2003)
27. Smirnov, S., Weidlich, M., Mendling, J.: Business process model abstraction based on synthesis from well-structured behavioral profiles. INT J COOP INF SYST 21(01), 55–83 (2012)
28. Solé, M., Carmona, J.: Process mining from a basis of state regions. In: Petri Nets. LNCS, vol. 6128, pp. 226–245. Springer (2010)
29. Weidlich, M., Polyvyanyy, A., Mendling, J., Weske, M.: Causal behavioural profiles - efficient computation, applications, and evaluation. Fundam. Inform. 113(3-4), 399–435 (2011)
30. Weidlich, M., van der Werf, J.M.E.M.: On profiles and footprints - relational semantics for Petri nets. In: Petri Nets. LNCS, vol. 7347, pp. 148–167. Springer (2012)
31. Weijters, A., van der Aalst, W., de Medeiros, A.: Process mining with the heuristics miner-algorithm. BETA Working Paper Series 166 (2006)
32. Weijters, A.J.M.M., Ribeiro, J.T.S.: Flexible heuristics miner. In: CIDM. pp. 310–317. IEEE (2011)
33. 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)
34. Wen, L., Wang, J., Sun, J.: Mining invisible tasks from event logs. Advances in Data and Web Management pp. 358–365 (2007)
35. van der Werf, J., van Dongen, B., Hurkens, C., Serebrenik, A.: Process discovery using integer linear programming. Fundamenta Informaticae 94, 387412 (2010)
36. Yzquierdo-Herrera, R., Silverio-Castro, R., Lazo-Cortés, M.: Sub-process discovery: Opportunities for process diagnostics. In: EIS of the Future, pp. 48–57. Springer (2013)

A SMT translation

A.1 $\times, \rightarrow, \wedge$

Cut searches for $\times$, $\rightarrow$ and $\wedge$ are translated straightforwardly to optimisation problems, by maximising the average probability of edges crossing the cut.

For $\oplus \in \{\times, \rightarrow, \wedge\}$, $\mathrm{p}_\oplus = \frac{\sum_{a_1 \in \Sigma_1, a_2 \in \Sigma_2} \mathrm{p}_\oplus(a_1, a_2)}{|\Sigma_1| \cdot |\Sigma_2|} = \frac{k}{l}$. The basic decision to be made by the SMT solver is how to divide the activities in two sets: the ones on one side of the cut ($cut(a)$) and the ones on the other side of the cut ($\neg cut(a)$), such that $\mathrm{p}_\oplus$ is maximised. Let n be $|\Sigma(L)|$. As divisions cannot be translated to SMT directly, we repeatedly check while varying l. For the commutative $\times$ and $\wedge$, $l \in 1 \ldots n/2$, for the non-commutative $\rightarrow$, $l \in 1 \ldots n-1$. We give the translation to SMT for non-commutative $\rightarrow$; the commutative $\times$ and $\wedge$ are similar.

Given an l, the number of nodes cut must be l:

$$|\{a|cut(a)\}| = l$$

$\mathrm{p}_{\oplus}$ is defined on pairs, so for each pair of activities (a_1, a_2) we introduce a helper variable $crosses(a_1, a_2)$, denoting whether (a_1, a_2) crosses the cut:

$$crosses(a_1, a_2) \Leftrightarrow (cut(a_1) \wedge \neg cut(a_2))$$

The objective function to be maximised is the weighted sum of the crossing edges:

$$obj = \sum_{a_1 \in \Sigma_1, a_2 \in \Sigma_2} crosses(a_1, a_2) \cdot \mathrm{p}_{\underline{\oplus}}(a_1, a_2)$$

Once an optimal solution is found, $\frac{obj}{l}$ gives the probability of $\mathrm{p}_{\oplus}$.
For $\wedge$, a constraint is added that both Σ_1 and Σ_2 contain both start and end activities.

A.2 $\circlearrowleft$

For $\mathrm{p}_{\circlearrowleft}$, each pair (a, b) that crosses the cut is categorised as being either indirect, single or reverse single. (The fourth category, double, cannot happen in class C_R and is therefore omitted here.) They are defined as follows:

$$\begin{aligned}
(a,b) \text{ single} &\Leftrightarrow (a \in End_1 \wedge b \in Start_2) \vee \\
&\quad\ (a \in End_2 \wedge b \in Start_1) \\
(a,b) \text{ reverse single} &\Leftrightarrow (b,a) \text{ single} \\
(a,b) \text{ indirect} &\Leftrightarrow \neg(a,b) \text{ single} \ \wedge \neg(b,a) \text{ single}
\end{aligned}$$

We give an example using Figure 10, showing a directly-follows graph. In this example, $\Sigma_1 = \{u, v\}$, $\Sigma_2 = \{w, x\}$, $Start_1 = \{u\}$, $End_1 = \{v\}$, $Start_2 = \{x\}$, and $End_2 = \{w\}$. Pairs (u, x), (v, w), (x, u) and (w, v) are indirect, (w, u) and (v, x) are single and (u, w) and (x, v) are reverse single. Note that indirect corresponds to $\underline{\circlearrowleft_{\mathrm{i}}}$ and single to $\underline{\circlearrowleft_{\mathrm{s}}}$.

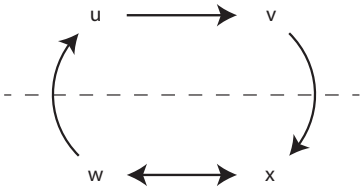


Figure 10: Example $\mapsto$-graph. The dashed line denotes a cut.

The optimisation searches for assignments to Σ_1, Σ_2, $Start_2$ and End_2, and a classification of all edges that maximises the average probability of single and indirect edges. $Start_1$ and End_1 are taken as-is from the log.

B Proof of Lemma 10

Proof. We follow a similar reasoning as in the proof of Lemma 9 to prove that $\mathrm{p}_{\oplus}(\Sigma_1, \Sigma_2) > \mathrm{p}_{\oplus}(\Sigma_1', \Sigma_2')$: we prove a lower bound for $\mathrm{p}_{\oplus}(\Sigma_1, \Sigma_2)$, an upper bound for $\mathrm{p}_{\oplus}(\Sigma_1', \Sigma_2')$ and compare these two. Apply case distinction on whether $\oplus = \circlearrowleft$.

– Case $\oplus \neq \circlearrowleft$. Obviously, (1) holds in this case as well. For the upper bound for $\mathrm{p}_{\oplus}(\Sigma_1', \Sigma_2')$, we start with

$$\frac{\sum_{a \in \Sigma_1', b \in \Sigma_2'} \mathrm{p}_{\underline{\oplus}}(a, b)}{|\Sigma_1'| \cdot |\Sigma_2'|} = \mathrm{p}_{\oplus}(\Sigma_1', \Sigma_2') \tag{4}$$

As $c' = (\oplus, \Sigma_1', \Sigma_2')$ does not conform to M, there is a $\Sigma(M_i)$ partitioned by c': $\Sigma_1' \cap \Sigma(M_i) \neq \varnothing$ and $\Sigma_2' \cap \Sigma(M_i) \neq \varnothing$. Consider this $M_i = \otimes(\ldots)$, then $c'_{M_i} = (\otimes, (\Sigma(M_i) \cap \Sigma_1'), (\Sigma(M_i) \cap \Sigma_2')$ is a cut of M_i. Take an arbitrary cut c_{M_i} that conforms to M_i. By Lemma 8, at least one activity pair (u, v) is partitioned by both c_{M_i} and c'_{M_i}. For all other $(a, b) \neq (u, v)$, by Table 1, it holds that $\mathrm{p}_{\underline{\oplus}}(a, b) \leqslant 1$, and there are $|\Sigma_1| \cdot |\Sigma_2| - 1$ of those pairs. Applying this to (4), we derive:

$$\frac{(|\Sigma_1'| \cdot |\Sigma_2'| - 1) \cdot 1 + 1 \cdot \mathrm{p}_{\underline{\oplus}}(u, v)}{|\Sigma_1'| \cdot |\Sigma_2'|} \geqslant \mathrm{p}_{\oplus}(\Sigma_1', \Sigma_2')$$

As c_{M_i} conforms to M_i and $\oplus \neq \circlearrowleft$, $\underline{\otimes}(u, v)$ holds. As M is in normal form $\otimes \neq \oplus$, and therefore $\underline{\oplus}(u, v)$ does not hold. Then, by Table 1, $\mathrm{p}_{\underline{\oplus}}(u, v) \leqslant \frac{1}{z(u,v)+1}$. From $z(u, v) = \frac{|u|+|v|}{2} \geqslant 1$ follows that $\mathrm{p}_{\underline{\oplus}}(u, v) \leqslant \frac{1}{z(u,v)+1} \leqslant \frac{1}{2}$. Define y to be $|\Sigma_1'| \cdot |\Sigma_2'|$.

$$\frac{(y-1) + \frac{1}{2}}{y} = \frac{(|\Sigma_1'| \cdot |\Sigma_2'| - 1) \cdot 1 + \frac{1}{2}}{|\Sigma_1'| \cdot |\Sigma_2'|} \geqslant \mathrm{p}_{\oplus}(\Sigma_1', \Sigma_2') \tag{5}$$

Similar to the proof of Lemma 9, from (1), (5) and choosing $k > 2y - 1$, follows that $\mathrm{p}_{\oplus}(\Sigma_1, \Sigma_2) > \mathrm{p}_{\oplus}(\Sigma_1', \Sigma_2')$.

– Case $\oplus = \circlearrowleft$. We follow a reasoning similar to the proof of Lemma 9, and derive the lower bound (1) again. For the upper bound for $\mathrm{p}_{\oplus}(\Sigma_1', \Sigma_2')$, similar to the proof of Lemma 9, we derive

$$\frac{\sum_{a \in \Sigma_1', b \in \Sigma_2'} \mathrm{p}_{\underline{\oplus}}(a, b)}{|\Sigma_1'| \cdot |\Sigma_2'|} \geqslant \mathrm{p}_{\oplus}(\Sigma_1', \Sigma_2')$$

As $L \diamond\!\!\rightarrow M$ and by semantics of $\circlearrowleft$, $\mapsto^+$ holds for all activity pairs. Thus, $\underline{\circlearrowleft_{\mathrm{s}}} \cup \underline{\circlearrowleft_{\mathrm{i}}} \cup \underline{\wedge}$ contains all activity pairs. By constraint, $Start(M) = Start(M_1) \subseteq \Sigma_1$ and $Start(M_1) \subseteq \Sigma_1'$.

c' separates at least a $\Sigma(M_i)$. Let (u, v) be a pair of activities of $\Sigma(M_i)$ separated by c'. Prove by case distinction on whether $\Sigma(M_i) = \Sigma_1$ that at least one pair (u, v) is counted wrongly.

- Case $\Sigma(M_i) = \Sigma_1$. Towards contradiction, assume no misclassified pair exists in $\Sigma(M_1)$. Take an arbitrary $a_k \in \Sigma(M_1)$. Apply case distinction on whether a_k is a start or an end activity.

  - If $a_k \in Start(M)$ or $a_k \in End(M)$, by constraint $a_k \in \Sigma_1'$.
  - Consider two $\mapsto$-paths: from a start activity to a_k: $a_1 \dots a_k$ such that $a_1 \in Start(M)$ and $a_{j>1} \notin Start(M) \cup End(M)$, and a $\mapsto$-path from a_k to an end activity: $a_k \dots a_l$ such that $a_l \in End(M)$ and $a_{j<l} \notin Start(M) \cup End(M)$. Apply case distinction on whether such paths exist.
    - $\exists a_1 \dots a_k$. Then some pair (a_p, a_q), on this path crosses c'. As (a_p, a_q) is on a $\mapsto$-path $a_p \mapsto a_q$, so either $\underline{\circlearrowleft_{\mathrm{s}}}(a_p, a_q)$ or $\underline{\asymp}(a_p, a_q)$. Activity a_q is not a start activity and a_p is not an end activity, so (a_p, a_q) contributes as $\underline{\circlearrowleft_{\mathrm{i}}}$ towards $\mathrm{p}_{\circlearrowleft}(c')$.
    - $\exists a_k \dots a_l$. Similar.
    - $\nexists a_1 \dots a_k \wedge \nexists a_k \dots a_l$. Then a_k must be on a $\mapsto$-path $a_l \dots a_k \dots a_1$ with $a_1 \in Start(M)$ and $a_k \in End(M)$. As $M_1 \neq \circlearrowleft$, this can only happen if $M_1 = \times$, which means that there is a $a_1' \in Start(M)$ such that no $\mapsto$-path $a_k \dots a_1'$ exists. Then, $\underline{\circlearrowleft_{\mathrm{i}}}(a_k, a_1')$, but (a_k, a_1') contributes as $\underline{\circlearrowleft_{\mathrm{s}}}$.
- Case $\Sigma(M_i) \neq \Sigma_1$. As M is reduced, $M_i \neq \oplus(\dots)$ and thus, the $\mapsto$-graph of M_i is connected. By semantics of process trees, there is at least a start or end activity that can be executed before/after both u and v: either $\langle s \cdots u \rangle$ and $\langle s \cdots v \rangle$ or $\langle u \cdots e \rangle$ and $\langle v \cdots e \rangle$, with $s \in Start(M_i)$ and $e \in End(M_i)$. Without loss of generality, assume that two $\mapsto$-paths $\langle s \cdots u \rangle$ and $\langle s \cdots v \rangle$ exist in the $\mapsto$-graph, such that $s \in Start(M_i)$. The pair (u, v) crosses c', so one of these paths must cross c' as well. Let (x, y) be such a crossing pair in the $\mapsto$-graph. As $\mapsto(x, y)$, either $\underline{\asymp}(x, y)$ or $\underline{\circlearrowleft_{\mathrm{s}}}(x, y)$. Neither x nor y are start or end activities of M, so the pair (x, y) contributes as $\underline{\circlearrowleft_{\mathrm{i}}}$ to the average $\mathrm{p}_{\circlearrowleft}(c')$.

The remaining part of this case is similar to the case $\oplus \neq \circlearrowleft$.

Hence, $\mathrm{p}_{\oplus}(\Sigma_1, \Sigma_2) > \mathrm{p}_{\oplus}(\Sigma_1', \Sigma_2')$. □