Simplifying Discovered Process Models in a Controlled Manner

Dirk Fahland, Wil M.P. van der Aalst

Eindhoven University of Technology, The Netherlands

Abstract

Process models discovered from a process log using process mining tend to be complex and have problems balancing between overfitting and underfitting. An overfitting model allows for too little behavior as it just permits the traces in the log and no other trace. An underfitting model allows for too much behavior as it permits traces that are significantly different from the behavior seen in the log. This paper presents a post-processing approach to simplify discovered process models while controlling the balance between overfitting and underfitting. The discovered process model, expressed in terms of a Petri net, is unfolded into a branching process using the event log. Subsequently, the resulting branching process is folded into a simpler process model capturing the desired behavior.

Keywords: process mining, model simplification, Petri nets, branching processes

1. Introduction

Information systems are becoming more and more intertwined with the operational processes they support. While supporting these processes, multitudes of events are recorded, cf. audit trails, database tables, transaction logs, data warehouses. The goal of *process mining* is to use such event data to extract process-related information. The most prominent problem of process mining is *process discovery*, that is, to automatically *discover* a process model by observing events recorded by some information system. The discovery of process models from event logs is a relevant, but also challenging, problem [1–3].

Input for process discovery is a collection of traces. Each trace describes the life-cycle of a process instance (often referred to as case). Output is a process model that is able to reproduce these traces. The automated discovery of process models based on event logs helps to jump-start process improvement efforts and provides an objective up-to-date process description. There are two other kinds

Email addresses: d.fahland@tue.nl (Dirk Fahland), w.m.p.v.d.aalst@tue.nl (Wil M.P. van der Aalst)

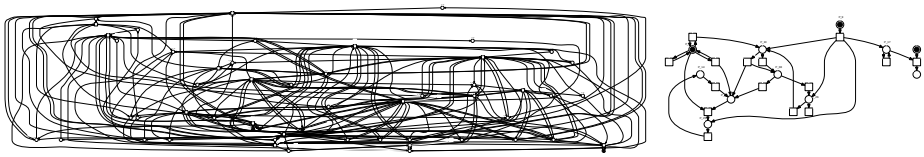

Figure 1: Hospital patient treatment process after process discovery (left) and after subsequent simplification using the approach presented in this paper (right).

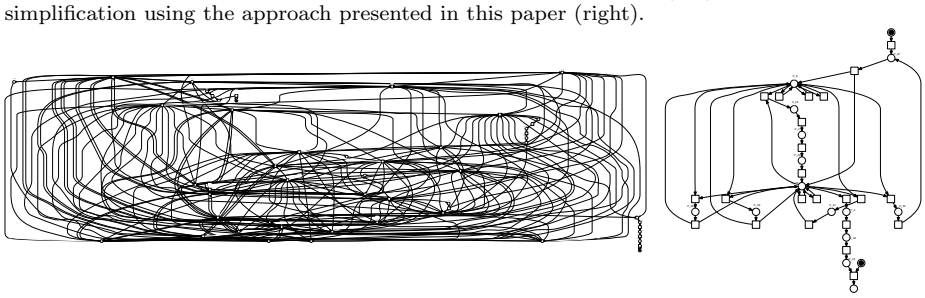

Figure 2: Municipality complaint process after process discovery (left) and after subsequent simplification (right).

of process mining. *Process extension* extends a given (handmade or discovered) process model with information from the log, for instance, by projecting a log on a discovered model to show bottlenecks and deviations. *Conformance checking* is the problem of measuring how well a handmade or discovered process model describes behavior in a given log [1].

The main problem of process discovery from event logs is to balance between *overfitting* and *underfitting*. A model is overfitting if it is too specific, i.e., the example behavior in the log is included, but new instances of the same process are likely to be excluded by the model. For instance, a process with 10 concurrent activities has $10! = 3628800$ potential interleavings. However, event logs typically contain fewer cases. Moreover, even if there are 3628800 cases in the log, it is extremely unlikely that all possible variations are present. Hence, an overfitting model (describing exactly these cases) will not capture the underlying process well. A model is underfitting when it over-generalizes the example behavior in the log, i.e., the model allows for behaviors very different from what was seen in the log. Process discovery is challenging because (1) the log typically only contains a *fraction* of all possible behaviors, (2) due to concurrency, loops, and choices the *search space has a complex structure*, and (3) there are *no negative examples* (i.e., a log shows what has happened but does not show what could not happen) [1, 3].

A variety of approaches has been proposed to address these challenges [1, 2]. Technically, all these approaches extract ordering constraints on activities which are then expressed as control-flow constructs in the resulting process model. Provided that enough event data are available and variability is low, today's approaches are able to discover the underlying process adequately. However, processes with more variability are more difficult to discover and numerous

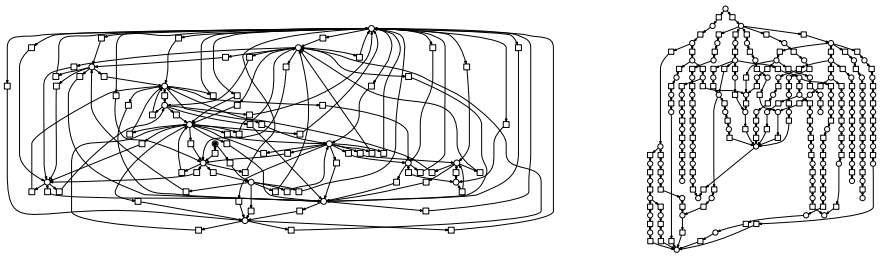

Figure 3: The hospital process (Fig. 1) discovered by [12] (left) can be simplified (right).

approaches have been proposed to deal with this.

Several approaches try to *abstract from infrequent behavior* and construct models that capture only the "highways" in processes. Examples are heuristic mining[1] [4], fuzzy mining [5], and genetic process mining [6]. The resulting models are relatively simple, but may not able to reproduce all traces seen in the log. These techniques exploit the fact that for many processes the so-called "80/20-rule" holds, i.e., 80% of the observed cases can be explained by 20% of the paths in the process whereas the remaining 20% of cases is responsible for 80% of the variability. Although the techniques proposed in [4–6] can simplify models, parts of the event log are no longer explained by the model and the model is often not executable because split-join behavior is either unspecified (fuzzy mining) or implicit (heuristic mining and genetic process mining) [7].

Other approaches try to deal with variability by *constructing an over-general model*. Instead of leaving out infrequent behavior, everything is allowed unless there is strong evidence that it is not possible. This can easily be understood in terms of a Petri net. A Petri net with transitions T and without any places can reproduce any event log over a set of activities T. Adding a place to a Petri net corresponds to adding a constraint on the behavior. Techniques based on *language-based regions* [8, 9] use this property. For example, as shown in [9] it is possible to solve a system of inequations to add places that do not inhibit behavior present in the event log. In [10], an approach based on convex polyhedra is proposed. Here the Parikh vector of each prefix in the log is seen as a polyhedron. By taking the convex hull of these convex polyhedra one obtains an over-approximation of the possible behavior. In [11], the authors resort to the use of OR-splits and OR-joins to create an over-general model that guarantees that all traces in the log can be reproduced by the model. Surprisingly, these over-general process models tend to be convoluted as illustrated by Fig. 1 and 2.

In [12] an approach to balance overfitting and underfitting is proposed. First,

[1]Historically, process discovery and process mining were used synonymously, as discovery was the first and most prominent process mining problem that was addressed. Thus, various discovery techniques are called "mining techniques" although they just focus on discovery. We will use their original name, but use the term "discovery" to refer to the problem.

a transition system is constructed from the log; the user may balance generalization by influencing how states are generated from the log. Then, a Petri net is derived from this transition system. The approach requires expert knowledge to specify the right abstraction that balances overfitting and underfitting. If applied correctly, this technique yields simpler models (compare Fig. 3 (left) and Fig. 1 (left)), but even these models are still convoluted and can be simplified as shown by Fig. 3 (right).

The problem that we address in this paper is to *structurally simplify* a mined process model N while *preserving that N can replay the entire log L from which it was generated*; a model is *simpler* if it shows less interconnectedness between its nodes, see Figs. 1-3. We propose a set of techniques for re-adjusting the generalization done by process discovery algorithms, and to cut down involved process logic to the logic that is *necessary* to explain the observed behavior. Note that our approach is *not* intended as a replacement for existing discovery techniques. It does not infer causal dependencies between activities. Instead, *it can be used as a post-optimization for any process discovery technique whose results can be represented as Petri nets.* Also, our approach is *not limited* to discovered process models: also a hand-made model can be "post-processed" with respect to a given log.

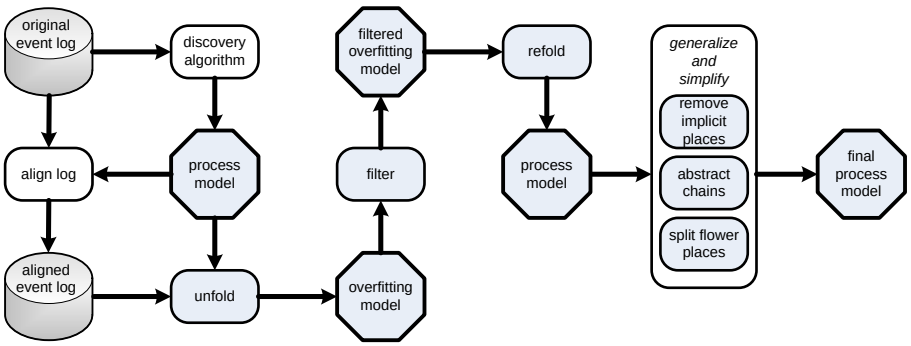

Figure 4: Overview of the approach to simplify mined process models.

Figure 4 shows the overall approach proposed in this paper. Starting point for our approach is an event log L and a discovered process model $N = \mathcal{M}(L)$. $\mathcal{M}$ is some conventional process discovery algorithm able to produce a Petri net, e.g., [8–10, 13–15]. Results by other algorithms such as heuristic mining [4] or genetic mining [6] can be converted into a Petri net as shown in [7, 16].

Some discovery algorithms $\mathcal{M}$ guarantee a fitting model, i.e., all traces in L can be replayed on N. However, using the approach described in [17, 18] we can align log and model when the discovery algorithm itself does not ensure this. The basic idea is that the non-fitting traces are massaged to fit the model. The resulting fitting log L' and the discovered model N are used to generate an *unfolded overfitting* model β. Technically speaking, we construct the so-called *branching process* of N only allowing for the observed behavior in the aligned

event log. The branching process also shows frequencies and can be used to remove infrequent behavior (if desired). The resulting filtered overfitting model can be *folded* into a process model. During folding the observed behavior is generalized in a controlled manner. After folding, we apply further generalization and simplification techniques. For example, we reduce superfluous control-flow structures by removing implicit places from the model and define abstraction operations to simplify the structure and generalize the described behavior in a disciplined manner.

Figures 1-3 illustrate the effectiveness of our approach. Interestingly, it can be combined with any of the existing process discovery techniques. Moreover, as Fig. 4 already suggests, the user can influence the result. This is important as only the user can decide on the degree of simplification and generalization needed. The whole approach is supported by the process mining toolkit *ProM* which can be downloaded from `www.processmining.org`. We validated the feasibility of our technique in a number of experiments to simplify benchmark processes as well as process models from industrial case studies.

In the remainder of this paper, we first introduce preliminaries regarding event logs, Petri nets, partially ordered runs, and branching processes (Sect. 2). The subsequent sections describe the different steps depicted in Fig. 4. Sect. 3 shows the steps to come to a (filtered)f overfitting model. Basically, the model is projected onto the behavior actually observed while addressing issues related to infrequent or non-fitting behavior. In Sect. 4 it is shown how refolding can be used to generalize behavior in a controlled manner. Sect. 5 defines the operations for further simplifying the folded model. We report on experimental results in Sect. 6. Sect. 7 discusses related work and Sect. 8 concludes the paper.

2. Causal Behavior of Process Models w.r.t. an Event Log

This section introduces the notion of an event log and recalls some standard notions from Petri net theory. In particular the notion of a *branching process* will be vital for our approach.

2.1. Event Logs

Process mining aims to discover, monitor and improve real processes by extracting knowledge from event logs available in today's information systems. Starting point for process discovery is an *event log*. Each event in such a log refers to an *activity* (i.e., a well-defined step in some process) and is related to a particular *case* (i.e., a *process instance*). The events belonging to a case are *ordered* and can be seen as one "run" of the process. Event logs may store additional event attributes. In fact, whenever possible, process mining techniques use attributes such as the *resource* (i.e., person or device) executing or initiating the activity, the *timestamp* of the event, or *data elements* recorded with the event (e.g., the size of an order).

In this paper we abstract from these additional attributes and focus on control-flow discovery. Each case is described by a *trace*, i.e., a sequence of

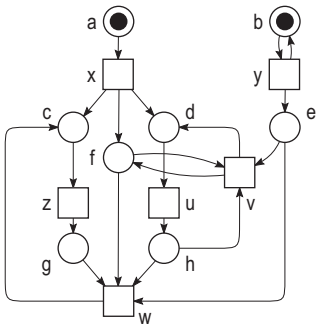

Figure 5: A net system N.

activity names. Different cases may have the same trace. Therefore, an event log is a *multiset* of traces (rather than a set).

Definition 1 (Event log)**.** Let $\mathcal{A}$ denotes some universe of *activities*, i.e., actions that can be recorded in a log. $l \in \mathcal{A}^*$ is a *trace*, i.e., a sequence of activities. $L \in \mathbb{B}(\mathcal{A}^*)$ is an *event log*, i.e., a multiset of traces. $\Sigma(L)$ is the set of activities used in L.

For example, $L = [\mathsf{xzy}, \mathsf{xzy}, \mathsf{yy}, \mathsf{yy}, \mathsf{yy}]$ is an event log with five cases, two cases follow trace xzy and three follow trace yy. $\Sigma(L) = \{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$. The fact that multiple cases have the same trace is important for process discovery. Frequencies are used as a basis for removing outliers and detecting incompleteness. Nevertheless, we will often refer to a log L as an ordinary set of traces. From the context, it will be clear whether L refers to $L = [\mathsf{xzy}, \mathsf{xzy}, \mathsf{yy}, \mathsf{yy}, \mathsf{yy}]$ or $L = \{\mathsf{xzy}, \mathsf{yy}\}$.

2.2. Petri Nets

A process discovery algorithm $\mathcal{M}$ returns for a log L a Petri net $N = \mathcal{M}(L)$. Ideally, N is able to reproduce the event log, i.e., elements of L correspond to occurrence sequences of N.

Definition 2 (Petri net)**.** A Petri net (P, T, F) consists of a set P of *places*, a set T of *transitions* disjoint from P, and a set of arcs $F \subseteq (P \times T) \cup (T \times P)$. A *marking* m of N assigns each place $p \in P$ a natural number $m(p)$ of *tokens*; technically, $m \in \mathbb{B}(P)$ is a bag of marked places of P. A *net system* $N = (P, T, F, m_0)$ is a Petri net (P, T, F) with an *initial* marking m_0.

We write ${}^\bullet y := \{x \mid (x, y) \in F\}$ and $y^\bullet := \{x \mid (y, x) \in F\}$ for the *pre-* and the *post-set* of y, respectively. Fig. 5 shows a slightly involved net system N with the initial marking $[\mathsf{a}, \mathsf{b}]$. N will serve as our running example as its structural properties are typical for results of a discovery algorithm.

The semantics of a net system N are typically given by a set of *sequential runs*. A transition t of N is *enabled* at a marking m of N iff $m(p) \geq 1$, for all $p \in {}^\bullet t$. If t is enabled at m, then t may *occur* in the step $m \xrightarrow{t} m_t$ of N

that reaches the *successor marking* m_t with $m_t(p) = m(p) - 1$ if $p \in {}^\bullet t \setminus t^\bullet$, $m_t(p) = m(p) + 1$ if $p \in t^\bullet \setminus {}^\bullet t$, and $m_t(p) = m(p)$ otherwise, for each place p of N. A sequential run of N is a sequence $m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} m_2 \ldots$ of steps $m_i \xrightarrow{t_{i+1}} m_{i+1}, i = 0, 1, 2, \ldots$ of N beginning in the initial marking m_0 of N. The sequence $t_1 t_2 \ldots$ is an *occurrence sequence* of N. For example, in the net N of Fig. 5 transitions x and y are enabled at the initial marking [a, b]; the occurrence of x results in marking [c, f, d, b] where z, u, and y are enabled; xzyuwyz is a possible occurrence sequence of N.

Occurrence sequences of a net system N correspond to traces in the event log. Whereas traces in the log have a clear begin and end, this is less clear for occurrence sequences. Occurrence sequences start in the initial marking, but in a net system termination is undefined. Therefore, it is sometimes useful to define a set of *final markings*. In this case, only occurrence sequences leading to a final marking correspond to traces in the event log. For example, WF-nets have a designated source and sink place to model the begin and end of the life-cycle of a process instance [1]. In this paper, we will not enforce such a structure and allow models such as the net N of Fig. 5.

2.3. Partially Ordered Runs and Branching Processes

In the following, we study the behavior of N in terms of its *partially ordered* runs [19]. We will use so-called *branching processes* [20] to represent sets of partially ordered runs, e.g., an event log will be represented as a branching process. We first illustrate the idea of a partially ordered run of N by an example and then define the branching processes of N.

Partially ordered runs. A partially ordered run π orders occurrences of transitions by a *partial order* — in contrast to a sequential run where occurrences are totally ordered. A partially ordered run π is again represented as a Petri net. Such a Petri net is *labeled* and, since it describes just one run of the process, the preset (postset) of a place contains at most one element. The net π_1 in Fig. 6 describes a partially ordered run of the net N of Fig. 5. A partially ordered run π of a net system N has the following properties:

- Each place of π is called *condition* and is labeled with a place of N, each transition of π is called an *event* and is labeled with a transition of N.
- A condition b of π with label p describes a *token* on p, the conditions of π with an empty pre-set describe the initial marking of N.
- An event e of π with label t describes an occurrence of transition t which consumes the tokens ${}^\bullet e$ from the places ${}^\bullet t$ and produces the tokens $e^\bullet$ on the places $t^\bullet$.

For example, event e_2 of π_1 in Fig. 6 describes an occurrence of y consuming token b_2 from place b and producing token b_5 on e and a *new* token b_6 on b. The events of π_1 are partially ordered: e_5 *depends on* e_2 whereas neither e_1 depends on e_2 nor e_2 on e_1. That is, e_1 and e_2 are *concurrent*. The partially

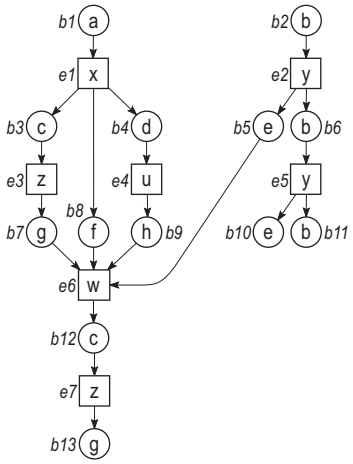

Figure 6: A partially ordered run π_1 of N of Fig. 5.

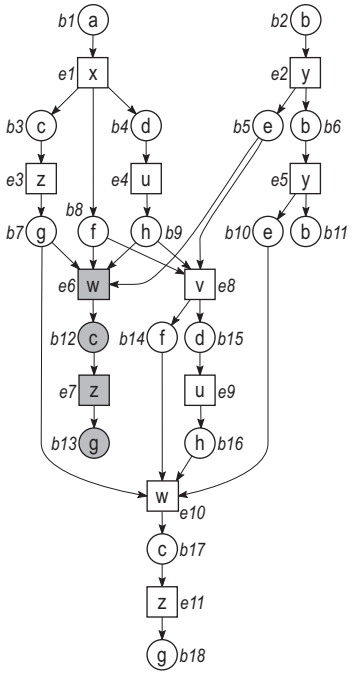

Figure 7: A branching process β the Petri net N of Fig. 5.

ordered run π_1 describes the occurrence sequence xzyuwyz — and several other sequences that order concurrent events differently such as yyxuzwz.

Branching processes. The partial order behavior of a net system N is the *set* of its partially ordered runs. A *branching process* represents a set of partially ordered runs in a single structure. This notion has been studied extensively in the last three decades [20–23] and we will use it to reason about the behavior of N.

A branching process β of N resembles an execution tree: each path of an execution tree denotes a run, all runs start in the same initial state, and whenever two runs diverge they never meet again. In β a "path" denotes a partially ordered run of N and we can read β as a special union of partially ordered runs of N: all runs start in the same initial marking, and whenever two runs diverge (by alternative events), they never meet again (each condition of β has at most one predecessor). Fig. 7 depicts an example of a branching process representing two partially ordered runs π_1 and π_2. π_1 is shown in Fig. 6, π_2 consists of the white nodes of Fig. 7. Both runs share b_1-b_{11} and e_1-e_5, and diverge at the alternative events e_6 and e_8 which compete for b_9 (i.e., a token in h) and also for b_8 and b_5.

A branching process of N is formally a *labeled* Petri net $\beta = (B, E, G, \lambda)$; each $b \in B$ ($e \in E$) is called *condition* (*event*), $\lambda \in (B \cup E) \to (P \cup T)$ assigns each node of β to a node of N such that $\lambda(b) \in P$ if $b \in B$ and $\lambda(e) \in T$ if $e \in E$.

Here, we give the constructive definition of the branching processes of N [22]. To begin with, we need some preliminary notions. Two nodes x_1, x_2 of β are in *causal relation*, written $x_1 \leq x_2$, iff there is path from x_1 to x_2 along the arcs G of β. x_1 and x_2 are in *conflict*, written $x_1 \# x_2$, iff there exists a condition $b \in B$ with distinct post-events $e_1, e_2 \in b^\bullet, e_1 \neq e_2$ and $e_1 \leq x_1$ and $e_2 \leq x_2$. x_1 and x_2 are *concurrent*, written $x_1 \,||\, x_2$ iff neither $x_1 \leq x_2$, nor $x_2 \leq x_1$, nor $x_1 \# x_2$. For example in Fig. 7 e_2 and e_9 are in causal relation ($e_2 \leq e_9$), e_7 and e_9 are in conflict ($e_7 \# e_9$), and e_3 and e_9 are concurrent ($e_3 \,||\, e_9$).

The branching processes of a Petri net $N = (P, T, F, m_0)$ are defined inductively:

Base. Let $B_0 := \bigcup_{p \in P} \{b_p^1, \ldots, b_p^k \mid m_0(p) = k\}$ be a set of conditions such that $\lambda(b_p^i) = p$ for $b_p^i \in B_0$. B_0 represents the *initial marking* of N. Then $\beta := (B_0, \emptyset, \emptyset, \lambda)$ is a branching process of N.

Assumption. Let $\beta = (B, E, G, \lambda)$ be a branching process of N. Let $t \in T$ with ${}^\bullet t = \{p_1, \ldots, p_k\}$. Let $\{b_1, \ldots, b_k\} \subseteq B$ be pair-wise concurrent conditions (i.e., $b_i \,||\, b_j$, for all $1 \leq i < j \leq k$) with $\lambda(b_i) = p_i$, for $i = 1, \ldots, k$. The conditions $b_1, \ldots, b_k$ together represent tokens in the pre-set of t.

Step. If there is no post-event e of $b_1, \ldots, b_k$ that represents an occurrence of t, then a new occurrence of t can be added to β. Formally, if there is no post-event $e \in \bigcap_{i=1}^k b_i{}^\bullet$ with $\lambda(e) = t$, then t is *enabled* at $\{b_1, \ldots, b_k\}$. We call $\{b_1, \ldots, b_k\}$ *enabling location* of t in β. Then the Petri net $\beta' = (B \cup C, E \cup \{e\}, G', \lambda')$ is

obtained from β by adding

- a fresh event e (not in β) with label $\lambda'(e) = t$ with ${}^\bullet e = \{b_1, \ldots, b_k\}$, and
- a fresh post-condition for each of the output places of t, i.e., for $t^\bullet = \{q_1, \ldots, q_m\}$, the set of conditions $C = \{c_1, \ldots, c_m\}$ is added to β' such that $C \cap B = \emptyset$ and for $i = 1, \ldots, m$: $\lambda'(c_i) = q_i$, ${}^\bullet c_i = \{e\}$.

β' is a branching process of N. For example, assume the branching process β of Fig. 7 without $e_{10}, b_{17}, e_{11}, b_{18}$ to be given. The conditions $\{b_7, b_{14}, b_{16}, b_{10}\}$ are pair-wise concurrent and represent tokens in ${}^\bullet w$ of N of Fig. 5. Appending e_{10} (labeled w) and b_{17} (labeled c) represents an occurrence of w; event e_{11} of z is added in the same way.

The arcs of a branching process β of N form a partial order, and any two nodes x_1 and x_2 are either in causal relation, in conflict, or concurrent [22]. Moreover, every Petri net N has a unique, possibly infinite, maximal branching process $\beta(N)$ which contains every other branching process of N as a prefix [20].

3. Reconsider Generalization: Create an Overfitting Model

Returning to our original problem setting, we now consider the behavior of a Petri net $N = \mathcal{M}(L)$ that was discovered from an event log L by some conventional discovery algorithm $\mathcal{M}$. In this section, we show how to construct an overfitting model that can be generalized and simplified in a controlled manner. Ideally, model N generated by the discovery algorithm is able to reproduce the log. However, in general, this does not need to be the case because of outliers (i.e., deliberate abstraction) or imperfections of the discovery technique. Therefore, we first align event log and model (Sect. 3.1). Then, we show how to create an overfitting model by restricting the branching process to actually observed behavior (Sect. 3.2). We can further restrict the branching process by removing infrequent behavior (Sect. 3.3).

3.1. Aligning Event Log and Model

Let $N = \mathcal{M}(L)$ be a net system discovered for an event log L while using algorithm $\mathcal{M}$. There are dozens of discovery algorithms that directly produce a net system [8–10, 12–15]. However, it is also possible to first discover a model using a different representation (e.g., [4–6]) and then convert it to a Petri net (cf. [7, 16]).

Some approaches, e.g., the techniques based on language-based regions [8, 9], guarantee that each trace $l \in L$ is an occurrence sequence of the discovered net system N. However, most algorithms will not guarantee that all traces of L fit perfectly. This implies that there may be a trace $l \in L$ that cannot be reproduced by N. Also N could be hand-made and not fit the log L. This is a problem for the techniques described in this paper. For example, it is not possible to unfold a net based on an event log that does not match the model. There are basically two ways to address this problem.

- Remove all non-fitting traces from L, i.e., l is removed from L if it is not an occurrence sequence of N.
- Massage the non-fitting traces such that all fit, i.e., if $l \in L$ is not an occurrence sequence of N, it is transformed into the "closest" occurrence sequence l'.

In most cases, the second approach is preferable. For larger processes with lots of variability there may be just a few cases in the log that fit the model from begin to end. It does not make sense to remove the majority of cases. Therefore, we elaborate on *aligning model and log*. Consider net N of Fig. 5 and $l = \mathsf{xywzu} \in L$. l does not fit N. Therefore, we consider alternative *alignments* such as:

$$\gamma_1 = \left|\begin{array}{c|c|c|c|c|c|} \mathsf{x} & \mathsf{y} & \mathsf{w} & \mathsf{z} & \mathsf{u} \\ \hline \mathsf{x} & \mathsf{y} & \bot & \mathsf{z} & \mathsf{u} \end{array}\right| \quad \text{and} \quad \gamma_2 = \left|\begin{array}{c|c|c|c|c|c|c|} \mathsf{x} & \mathsf{y} & \bot & \bot & \mathsf{w} & \mathsf{z} & \mathsf{u} \\ \hline \mathsf{x} & \mathsf{y} & \mathsf{z} & \mathsf{u} & \mathsf{w} & \mathsf{z} & \bot \end{array}\right|$$

Alignment γ_1 aligns trace $l = \mathsf{xywzu}$ with occurrence sequence xyzu. Ideally event log and model make the same "moves". For example, the first two moves of trace l can be mimicked by N. However, move w in the log cannot be followed by the model. Therefore, there is a "move in log only", denoted $(\mathsf{w}, \bot)$, in the third position of alignment γ_1. The next two positions in γ_1 show that after this, event log and model can make identical moves again. Alignment γ_2 aligns trace l with occurrence sequence xyzuwz. Again model and log "agree" on the first two moves. This is followed by two "moves in model only" ($(\bot, \mathsf{z})$ and $(\bot, \mathsf{u})$), i.e., in the model z and u occur without a matching move in the log. Then model and log "agree" on the next two moves. However, as position seven in the alignment shows, there is a "move in log only", denoted $(\mathsf{u}, \bot)$, because u is not enabled. Given a trace like l there are many possible alignments. However, as shown in [17, 18] it is possible to associate costs to the different "moves" and select an "optimal" alignment. For example, when assuming unit costs for all moves where model and log disagree, γ_1 (cost 1) is a better alignment than γ_2 (cost 3).

By selecting optimal alignments and replacing each trace of L with its "optimally aligned" occurrence sequence, we can convert log L into a log L' such that any $l \in L'$ is an occurrence sequence of N. This allows us to *only consider perfectly fitting logs independent of the discovery algorithm used*. In the remainder, L will always refer to the log *after* alignment.

3.2. Branching Process Restricted to Observed Behavior

The maximal branching process $\beta(N)$ of N introduced in Sect. 2.3 describes all behavior of N, not only the cases recorded in L. This additional behavior was introduced by the discovery algorithm $\mathcal{M}$ which discovered N from L. To re-adjust the generalization, we restrict the behavior $\beta(N)$ to L and derive an *overfitting* process model $N(L)$ that exhibits exactly L.

The restriction of $\beta(N)$ to the cases L is the branching process $\beta(L)$ that we obtain by restricting the inductive definition of the branching processes of N to

the cases in L. Beginning with $\beta = (B_0, \emptyset, \emptyset, \lambda_0)$, iterate the following steps for each case $l = t_1 t_2 \dots t_n \in L$. Initially, let $M := B_0$, $i := 1$.

1. Let $\{p_1, \dots, p_k\} = {}^\bullet t_i$.
2. If there exists $\{b_1, \dots, b_k\} \subseteq M$ with $\lambda(b_j) = p_j$, $j = 1, \dots, k$ and $e \in \bigcap_{j=1}^{k} b_j{}^\bullet$ with $\lambda(e) = t_i$, then $M := (M \setminus {}^\bullet e) \cup e^\bullet$.
   [The occurrence e of t_i is already represented at $\{b_1, \dots, b_k\}$; compute the successor marking of M by consuming the tokens ${}^\bullet e$ from the pre-places ${}^\bullet t_i$ and producing the tokens $e^\bullet$ on $t_i{}^\bullet$.]
3. Otherwise, choose $\{b_1, \dots, b_k\} \subseteq M$ with $\lambda(b_j) = p_j$, $j = 1, \dots, k$, and append a new event $e, \lambda(e) = t_i$ to all $b_j, j = 1, \dots, k$, and append a new condition c to e (with $\lambda(c) = q$) for each $q \in t^\bullet$. $M := (M \setminus {}^\bullet e) \cup e^\bullet$.
   [Add a new occurrence e of t_i at $\{b_1, \dots, b_k\}$ and compute the successor marking.]
4. $i := i + 1$, and return to step 1 if $i \leq n$.

This procedure replays each $l \in L$. This is possible because l is also an occurrence sequence of N. If not, use the preprocessing step described in Sect. 3.1. By construction, $\beta(L)$ is a smallest prefix of $\beta(N)$ that represents each $l \in L$. We call $\beta(L)$ the *L-induced unfolding* of N. Step 3 is non-deterministic when marking M puts more than one token on a place. The results in this paper were obtained by treating M as a queue: the token that is produced first is also consumed first.

For example, the branching process of Fig. 7 is the branching process $\beta = \beta(L)$ of net N of Fig. 5 for the log $L = [\mathsf{xzuywz}, \mathsf{xzuyvuywz}, \mathsf{xzyuwz}, \mathsf{xyzuvuywz}, \mathsf{xuzywz}, \mathsf{xuzyywz}, \mathsf{yyxuvuzwz}, \dots]$.

$\beta(L)$ not only succinctly represents L, *but also all cases that differ from L by reordering concurrent actions.* The discovery algorithm that returned N determines whether two actions are concurrent. Further, $\beta(L)$ already defines a Petri net that exhibits the log L. By putting a token on each minimal condition b of $\beta(L)$ with ${}^\bullet b = \emptyset$, we obtain a labeled Petri net $N(L) = (B, E, G, \lambda, m_0)$ that exhibits exactly $\beta(L)$, i.e., $N(L)$ restricts the behavior of N to L.

3.3. Removing Infrequent Behavior From the Branching Process

The procedure just described constructs $N(L)$ without considering the frequencies of traces in L. Whether a trace appears once or many times will yield the same overfitting process model. However, it is easy to assign counters to all events when constructing the branching process restricted to log L.

Formally, we create a counter $\kappa \in E \to \mathbb{N}$. In Step 2 of the procedure described in Section 3.2, we increment the counter as follows $\kappa(e) := \kappa(e) + 1$. In Step 3 of the procedure we initialize the counter $\kappa(e) := 1$. After constructing $\beta(L)$, counter κ indicates how often an event was executed in the branching process after replaying the entire log. Note that the frequencies of traces matter for this counter. Therefore, we defined an event log to be a multiset rather than a set.

To illustrate function κ consider the event log $L = [\mathsf{xzuywz}^{20}, \mathsf{xzuyvuywz}^{30}, \mathsf{xzyuwz}^{15}]$. The superscripts indicate how frequent traces appear in the event

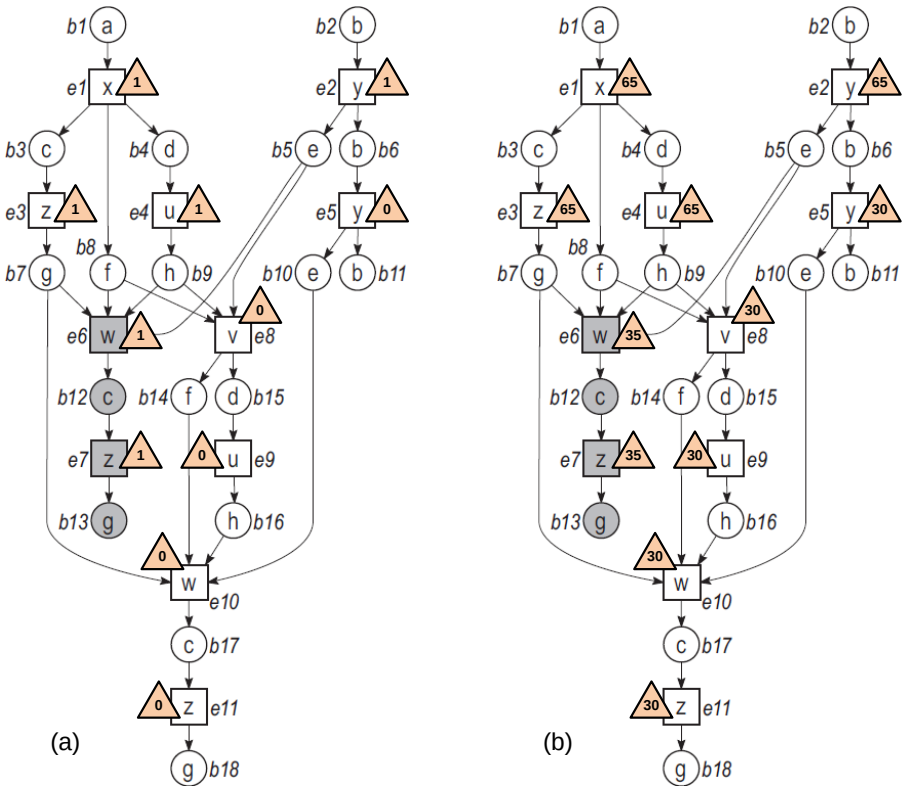

Figure 8: Counter κ: (a) after processing the first trace xzuywz and (b) after processing the whole event log $L = [\text{xzuywz}^{20}, \text{xzuyvuywz}^{30}, \text{xzyuwz}^{15}]$.

log. Log L contains 65 cases and trace xzuywz was observed 20 times. Applying the procedure described in Section 3.2 to this L will result in the branching process of Fig. 7. Let us first apply the procedure and update κ for only one trace: xzuywz. This results in the κ shown in Fig. 8(a), e.g., $\kappa(e_6) = 1$ and $\kappa(e_8) = 0$. After processing all 65 cases we get the κ shown in Fig. 8(b). Event e_1 is executed for all cases, therefore, $\kappa(e_1) = 65$. Event e_6 is executed for the 20 cases following xzuywz and the 15 cases following xzyuwz, hence, $\kappa(e_6) = 35$.

Function κ can be extended to conditions. For a condition $b \in B_0$, $\kappa(b) := |L|$. All other conditions b have precisely one pre-event e and $\kappa(b) := \kappa(e)$. This means that, while replaying the event log, condition b was marked $\kappa(b)$ times. κ values tend to decrease towards the leaves of the branching process. If two nodes x_1, x_2 are causally related, i.e., $x_1 \leq x_2$, then by definition $\kappa(x_1) \geq \kappa(x_2)$.

Function κ can be used to prune the overfitting model $\beta(L)$. We distinguish two possible reasons for removing an event e from $\beta(L)$.

- Event e is removed because $\kappa(e) < \tau_1$. Here τ_1 is an *absolute threshold*.

For example, for $\tau_1 = 33$, events e_5, e_8, e_9, e_{10}, and e_{11} will be removed from $\beta(L)$.

- Event e is removed because e is a post-event of some condition b and $\kappa(e)/\kappa(b) < \tau_2$. Here τ_2 is an *relative threshold*. Consider for example $\tau_2 = 0.50$. Based on this threshold, event e_8 would be removed because $\kappa(e_8)/\kappa(b_5) = 30/65 = 0.46 < 0.5$, i.e., of the 65 tokens produced for b_5 only 30 (i.e., 46%) were consumed by e_8.

It is also possible to use a mixture of both or to take into account the distance to the initial conditions. When removing an event e, also all causally dependent nodes $\{x \in B \cup E \mid e \leq x\}$ need to be removed.

Note that the representation shown in Fig. 8(b) can also be used to manually prune the overfitting branching process. In fact, a direct inspection of κ can provide interesting insights that cannot be obtained when looking at the log. Note, for example, that traces xzuywz and xzyuwz are obviously different when directly inspecting the event log. However, from the viewpoint of the branching process $\beta(L)$ and function κ, these two traces are equivalent (modulo reordering concurrent activities).

The overfitting model $\beta(L)$ and corresponding function κ can be used to remove infrequent behavior also referred to as "noise" or "outliers". When removing events from $\beta(L)$ the non-fitting parts of the event log need to be removed as they have been classified as being too infrequent.

In the remainder, we will not consider frequencies and assume an L, N, $\beta(L)$, and $N(L)$ such that each trace $l \in L$ is an occurrence sequence of both the original net N and the overfitting process model $N(L)$ (which in turn is based on $\beta(L)$). However, as demonstrated, our overall approach can deal with non-fitting and infrequent traces.

4. Generalize an Overfitting Model by Folding

The algorithm described in the preceding section yields for a Petri net N discovered from a log L, an overfitting net $N(L)$ that exhibits exactly the (filtered) branching process $\beta(L)$, i.e., the cases L (modulo reordering of concurrent actions). In the following, we present our main contribution: a number of operations that generalize $N(L)$ (introduce more behavior) and simplify the structure compared to N. Each operation addresses generalization and simplification in a different way and is independent of the other operations. So, a user may balance between the overfitting model $N(L)$ and the complex model N by choosing from the available generalization and simplification operations. We provide three kinds of operations which are typically executed in the given order.

1. $N(L)$ describes the cases L in an explicit form, i.e., only observed behavior is captured. We *fold* $N(L)$ to a more compact Petri net by identifying loops, and by merging similar behavior after an alternative choice. This

partly generalizes behavior of $N(L)$; the folded net is as most as complex as N (and typically much simpler).
2. Then we further simplify the folded net by removing *implicit* places. An implicit place does not constrain the enabling of transitions and hence can be removed [24]. Repeatedly removing implicit places can significantly simplify the net.
3. Finally, the net may have specific structures such as chains of actions of the same kind or places with a large number of incoming and outgoing arcs. We provide techniques to replace such structures by simpler structures. This allows us to generalize the behavior of $N(L)$ in a controlled way.

The *structural complexity* of N is its *simple graph complexity* $c(N) = \frac{|F|}{|P|+|T|}$ which correlates with the perceived complexity of the net, e.g., the complexities in Fig. 1 are 4.01 (left) and 1.46 (right). Each mentioned operation transforms a net N' into a net N'' guaranteeing that (1) N' exhibits at least each case of N'' (generalization), and (2) $c(N'') \leq c(N')$ (simplification).

This section describes the folding of $N(L)$; removing implicit places and other structural simplifications are explained in Sect. 5.

4.1. Folding an Overfitting Model

Our first step in creating a simplified process model is to *fold* the overfitting net $N(L)$ to a Petri net $N_f(L)$. $N_f(L)$ exhibits more behavior than $N(L)$ (generalization) and has a simpler structure than the original net N.

Technically, we fold the underlying branching process $\beta(L) = (B, E, G, \lambda)$ of $N(L)$ by an *equivalence relation* $\sim$ on $B \cup E$ that preserves labels of nodes, and the local environments of events. We write $\langle x \rangle_\sim := \{x' \mid x \sim x'\}$ for the equivalence class of node x. $\langle X \rangle_\sim = \{\langle x \rangle_\sim \mid x \in X\}$ is a set of equivalence classes.

Definition 3 (Folding equivalence)**.** Let β be a branching process of N. An equivalence relation $\sim$ on the nodes of β is a *folding equivalence* iff

1. $x_1 \sim x_2$ implies $\lambda(x_1) = \lambda(x_2)$, for all nodes x_1, x_2 of β, and
2. $e_1 \sim e_2$ implies $\langle {}^\bullet e_1 \rangle_\sim = \langle {}^\bullet e_2 \rangle_\sim$ and $\langle e_1 {}^\bullet \rangle_\sim = \langle e_2 {}^\bullet \rangle_\sim$, for all events e_1, e_2 of β.

Trivial folding equivalences are (1) the identity, and (2) the equivalence induced by the labeling λ: $x_1 \sim x_2$ iff $\lambda(x_1) = \lambda(x_2)$. Sect. 4.2 will present a folding equivalence tailored towards process discovery. Every folding equivalence of a branching process β induces a folded Petri net which is in principle the quotient of β under $\sim$.

Definition 4 (Folded Petri net)**.** Let β be a branching process of N, let $\sim$ be a folding equivalence of β. The *folded Petri net* (w.r.t. $\sim$) is $\beta_\sim := (P_\sim, T_\sim, F_\sim, m_\sim)$ where $P_\sim := \{\langle b \rangle_\sim \mid b \in B_\beta\}$, $T_\sim := \{\langle e \rangle_\sim \mid e \in E_\beta\}$, $F_\sim := \{(\langle x \rangle_\sim, \langle y \rangle_\sim) \mid (x, y) \in F_\beta\}$, and $m_\sim(\langle b \rangle_\sim) := |\{b' \in \langle b \rangle_\sim \mid {}^\bullet b' = \emptyset\}|$, for all $b \in B_\beta$.

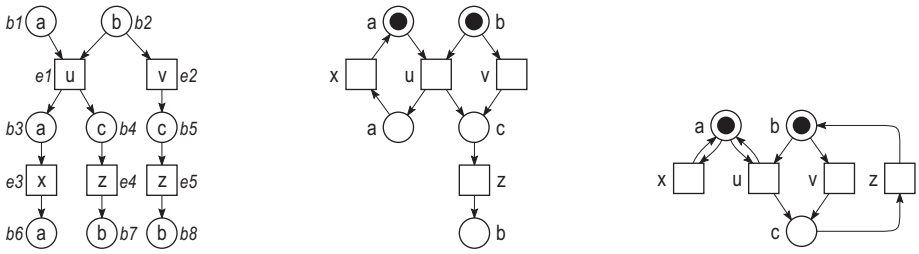

Figure 9: The branching process β^2 (left) can be folded to different nets N_2 (middle) and N_2' (right) using different folding equivalences.

For example, on β^2 of Fig. 9 we can define a folding equivalence $b_6 \sim b_1$, $b_4 \sim b_5$, $e_4 \sim e_5$, , $b_7 \sim b_8$ (and each node equivalent to itself). The corresponding folded net $\beta^2_\sim$ is N_2 of Fig. 9. The coarser folding equivalence defined by the labeling λ, i.e., $x \sim y$ iff $\lambda(x) = \lambda(y)$, yields the net N_2' of Fig. 9 (right). This example indicates that choosing a finer equivalence than the labeling equivalence yields a more explicit process model. Regardless of its explicitness, each folded net exhibits at least the original behavior $\beta(L)$.

Lemma 1. *Let N be a Petri net. Let β be a branching process of N with a folding equivalence $\sim$. Let $N_2 := \beta_\sim$ be the folded Petri net of β w.r.t. $\sim$. Then the maximal branching process $\beta(N_2)$ contains β as a prefix.*

Proof (Sketch). By Def. 3, all nodes of N_2 carry the same label, and the pre-set (post-set) of each transition t of N_2 is isomorphic to the pre-set (post-set) of each event of β defining t. Thus, $\beta(N_2)$ is built from the same events as β. By induction follows that N_2 can mimic the construction of β: for each event e with post-set that is added when constructing β, the transition $t = \langle e \rangle_\sim$ of N_2 leads to an isomorphic event e_2 that is added when constructing $\beta(N_2)$. Thus, we can reconstruct β (up to isomorphism) in $\beta(N_2)$. N_2 may allow to add more events to $\beta(N_2)$ than represented in β. These additional events are always appended to β, so β is a prefix of $\beta(N_2)$. See [25] for the full formal proof. □

4.2. The Future Equivalence

The following procedure *future*(β) constructs a folding equivalence (Def. 3) that specifically suits the simplification of discovered process models. The principle idea is to make all conditions that represent a token on a final place of the process model N (i.e., with an empty post-set) equivalent, and then to extend the equivalence as much as possible. To this end, we assume β to be finite which is the case for $\beta(L)$ introduced in Sect. 3.2.

1. Begin with the identity $x_1 \sim x_2$ iff $x_1 = x_2$, for all nodes x_1, x_2 of β.
2. While $\sim$ changes:
   for any two conditions b_1, b_2 of β with $\lambda(b_1) = \lambda(b_2)$ and ${b_1}^\bullet = {b_2}^\bullet = \emptyset$, set $b_1 \sim b_2$.

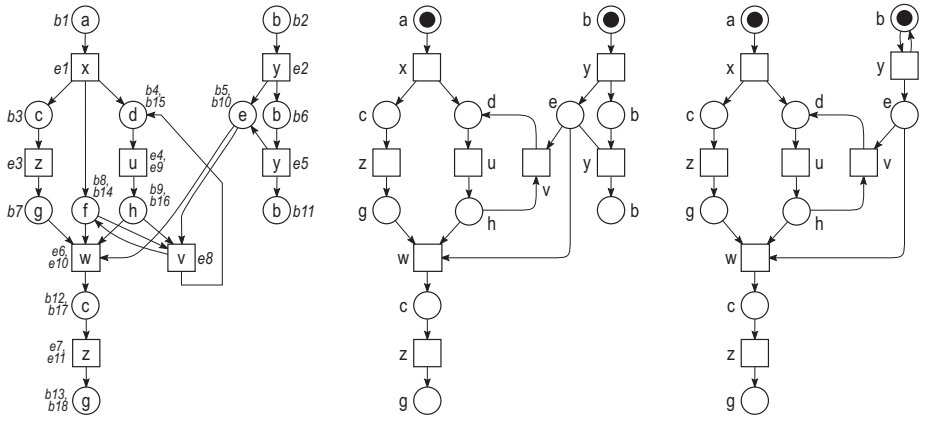

Figure 10: Folding the branching process β of Fig. 7 by *future*(β) yields the Petri net $N_f(\beta)$ (left). Removing places of the implicit conditions b_8 and b_{14} yields the Petri net $N_i(\beta)$ (middle). Abstracting the chain of y transitions yields the net $N_c(\beta)$ (right).

3. While $\sim$ changes:
   for any two events e_1, e_2 of β with $\lambda(e_1) = \lambda(e_2)$ and ${e_1}^\bullet = \{y_1, \ldots, y_k\}$, ${e_2}^\bullet = \{z_1, \ldots, z_k\}$ with $y_i \sim z_i$, for $i = 1, \ldots, k$, set $e_1 \sim e_2$, and set $u \sim v$, for any two pre-conditions $u \in {}^\bullet e_1$, $v \in {}^\bullet e_2$ with the same label $\lambda(u) = \lambda(v)$.
4. Return *future*(β) $:= \sim$.

Folding β along $\sim$ = *future*(β) merges the maximal conditions of β, i.e., rebuilds the final places of the process model of N, and then winds up β backwards as much as possible. This way, we also identify loops in the process model as illustrated in Fig. 10.

Taking β of Fig. 7 as input, the algorithm sets $b_{13} \sim b_{18}$ in step 2, b_{11} remains singleton. In the third step, first $e_7 \sim e_{11}$ and $b_{12} \sim b_{17}$ are set because of $b_{13} \sim b_{18}$; then $e_6 \sim e_{10}$ and $b_7 \sim b_7$, $b_8 \sim b_{14}$, $b_9 \sim b_{16}, b_5 \sim b_{10}$. The equivalence $b_9 \sim b_{16}$ introduces a loop in the folded model. Step 3 continues with $e_4 \sim e_9$ and $b_4 \sim b_{15}$, so that e_8 (v) has now b_4 (d) in the post-set. Folding β by this equivalence yields the net $N_f(\beta)$ of Fig. 10. It differs from N of Fig. 5 primarily in representing action z twice in different contexts. This example illustrates the main effect of *future*(β): to make process flow w.r.t. termination more explicit.

Complexitywise, *future*(β) has at most $|E|$ steps where events are merged; merging e_1 with another event requires to check at most $|E|$ events e_2; whether e_1 and e_2 are merged depends on the equivalence classes of their post-sets. Hence, *future*(β) runs in $\mathcal{O}(|E|^2 \cdot k)$ where k is the size of the largest post-set of an event.

The folded model $\beta_\sim$ exhibits the behavior β and possibly additional behavior. Some of this additional behavior may be problematic: if the original model N reaches an unsafe marking (i.e., a place has more than two tokens), the

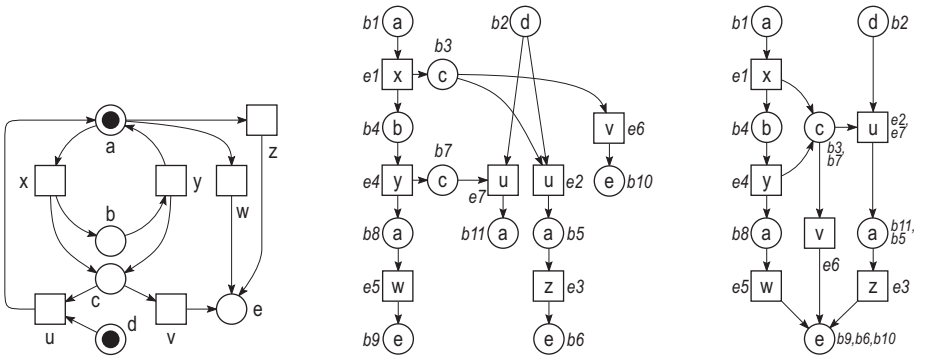

Figure 11: The unsafe net N^3 (left) has among others the non-deterministic branching process β^3 (middle); a deterministic future equivalence merges transitions and results in a deterministic net $N_d(\beta^3) = \beta^3_{det(future(\beta^3))}$ (right).

folded model $N_f(\beta) = \beta_\sim$ may reach a corresponding marking which enables two transitions $t_1 \neq t_2$ with the same label $a \in \Sigma(L)$. However, when replaying $l \in L$ one can select the wrong transition, potentially resulting in a deadlock. Fig. 11 illustrates the situation.

The net N^3 of Fig. 11 and the log $L^3 = \{\mathsf{xuzyw}, \mathsf{xyvuw}, \mathsf{xyuzw}\}$ yield the branching process $\beta^3 = \beta(N^3)$ shown in the middle. The future equivalence $\mathit{future}(\beta^3)$ would only join $b_9 \sim b_6 \sim b_{10}$. When replaying the third case xyuzw in β^3, we have to choose whether e_7 or e_2 shall occur; the choice determines whether the net can complete the case with z or ends in a deadlock.

We can solve the problem by *determinizing* the equivalence $\sim$ using the following procedure $\mathit{det}(\sim)$:

while $\sim$ changes do, for any two events e_1, e_2 of β, $e_1 \not\sim e_2$ with $\lambda(e_1) = \lambda(e_2)$, if there exist conditions $b_1 \sim b_2$ with $b_1 \in {}^\bullet e_1, b_2 \in {}^\bullet e_2$, then

1. set $e_1 \sim e_2$,
2. set $c_1 \sim c_2$, for all $c_1 \in {}^\bullet e_1, c_2 \in {}^\bullet e_2$ with $\lambda(c_1) = \lambda(c_2)$,
3. set $c_1 \sim c_2$, for all $c_1 \in e_1{}^\bullet, c_2 \in e_2{}^\bullet$ with $\lambda(c_1) = \lambda(c_2)$.

The resulting equivalence relation $\mathit{det}(\mathit{future}(\beta))$ is a folding equivalence that is coarser than the trivial equivalence defined by the identity on β and finer than the equivalence defined by the labeling of β. For example, determinizing $\mathit{future}(\beta^3)$ of Fig. 11 sets additionally $b_7 \sim b_3$, $e_7 \sim e_2$, $b_{11} \sim b_5$. Note that we can merge the two u labeled events because $b_2 \sim b_2$, $b_2 \in {}^\bullet e_7$, and $b_2 \in {}^\bullet e_2$ The resulting folded net $N_d(\beta^3) = \beta^3_{det(future(\beta^3))}$ of Fig. 11 (right) is indeed deterministic and can replay the entire log L^3.

The folded net $\beta_{det(future(\beta))}$ exhibits β (by Lem. 1) and possibly more behavior because the folding infers loops from β and merges nondeterministic transitions, which merges branches of unobservable choices until an observable choice. For our running example (β in Fig. 7), $N_f(\beta)$ is already deterministic (cf. Fig. 10), i.e., $N_d(\beta) = N_f(\beta)$.

The preceding operations of unfolding a mined net N to its log-induced branching process $\beta(L)$ and refolding $\beta(L)$ to $N_d(L) := \beta(L)_{det(future(\beta(L)))}$ yields a net that can replay all cases in L (by Lem. 1). The structure of $N_d(L)$ is at most as complex as the structure of the original net N — when $\beta(L)$ completely folds back to N. We observed in experiments that this operation typically reduces the complexity of N by up to 30%.

5. Controlled Generalization and Simplification

In Section 4 we showed how to fold a (filtered) log-inducted branching process $\beta(L)$ back to a Petri net N_d that exhibits $\beta(L)$ (and possibly more behavior). N_d may still be structurally complex. We can further simplify N_d with different techniques that we present in this section: (1) remove implicit places, (2) abstract a complex sub-structures to a more simple sub-structure, and (3) split a complex sub-structure into several simpler structures. We show in the following how to remove, abstract, or split the "right" places and sub-structures, so that the behavior of N_d is preserved or generalized in a controlled way.

5.1. Removing Implicit Places

A standard technique for structurally simplifying a Petri net N while *preserving* its behavior is to remove places that do not restrict the enabling of a transition. Such places are called *implicit*.

Definition 5 (Implicit place)**.** Let N be a Petri net. A pre-place p of a transition t of N is *implicit* iff whenever N reaches a marking m with tokens on each pre-place ${}^\bullet t \setminus \{p\}$, then also p has a token.

In other words, whether t is enabled only depends on ${}^\bullet t \setminus \{p\}$. Removing an implicit place p from N preserves the behavior of N up to tokens on p [24]. In the running example of Fig. 5, place f is implicit. Removing an implicit place p from N reduces its structural complexity because at least two adjacent arcs are removed as well. This yields an idea for our first simplification operation: remove as many implicit places from the folded net $N_d(\beta)$ as possible.

Note however that not all implicit places of a net can be removed, as an implicit place p can cease to be implicit when another implicit place gets removed. Finding a maximal set of implicit places is a well-known problem that can be solved by solving a system of linear (in-)equations [24] using an ILP-solver.

5.2. Removing Places Based on Implicit Conditions

We found in initial experiments that Petri nets discovered by process discovery algorithms contain only few implicit places according to Def. 5; the model simplification is only marginal. For example the model of Fig. 12(top left) contains no implicit place.

Fortunately, Def. 5 requires more than we do in our setting. The refolded net $N_d(\beta)$ which we want to simplify generalizes the behavior recorded in the log L to the occurrence sequences of $N_d(\beta)$. Removing implicit places of $N_d(\beta)$

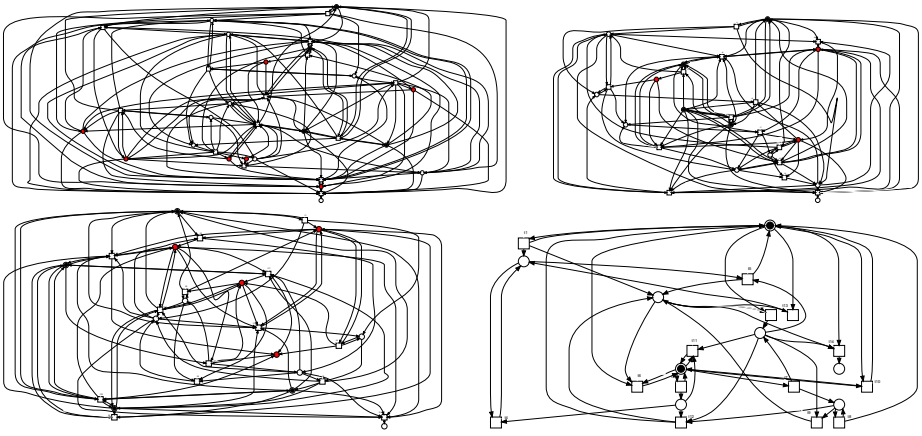

Figure 12: The net at the top left has no implicit conditions; it can be simplified by removing different variants of places based on implicit conditions.

preserves *all occurrence sequences* of $N_d(\beta)$ whereas we are only interested in preserving L, or more precisely, β. In the following, we introduce a family of techniques to simplify $N_d(\beta)$ by removing places based on *implicit conditions* of the log-induced unfolding β; the results of the different techniques are shown in Fig. 12.

5.2.1. Implicit conditions

By focusing on the L-induced unfolding β, we can discover which places in $N_d(\beta)$ can be removed while preserving the behavior L. The basic idea is to consider the *implicit conditions* of the net β. Removing any implicit condition b of β preserves the behavior of β (up to b), and hence preserves L.

Definition 6 (Implicit conditions)**.** Let β be a branching process. Let $imp(\beta)$ be the set of all implicit places of β according to Def. 5. We call a set $B' \subseteq imp(\beta)$ a *consistent* subset of implicit conditions of β iff for each $b \in B'$: b is implicit in β without $B' \setminus \{b\}$.

In contrast to implicit places, an implicit condition can easily be detected on the structure of β: a condition b with pre-event $\{e\} = {}^\bullet b$ is implicit iff for each post-event $f \in b^\bullet$ there exists a path from e to f that does not contain b. For example in Fig. 7, conditions b_8 and b_{14} are implicit.

Consider two implicit conditions $b_1, b_2 \in imp(\beta)$. After removing b_1 it may be that b_2 is no longer implicit. Therefore, we define the notion of a *consistent* subset of implicit conditions in Def. 6. All places in a consistent subset can be removed without changing the behavior whereas the removal of $imp(\beta)$ may enable more behavior.

Recall that the places in a folded Petri net have identifiers corresponding to equivalence classes of conditions in the branching process (cf. Def 4). Let $p = \{b_1, \ldots, b_k\}$ be such a place obtained using folding relation $\sim$. Removing

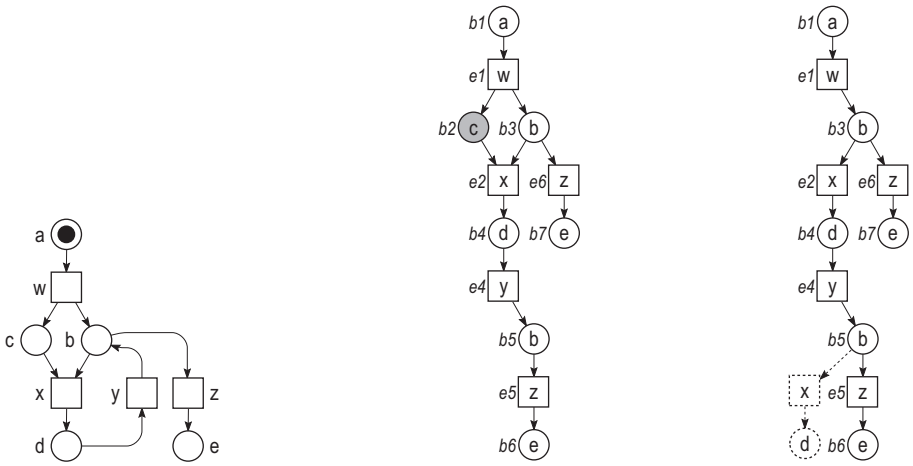

Figure 13: The net N (left) has no implicit places; the c-labeled condition b_2 of its log-induced unfolding β (middle) is implicit; removing c from N allows for more behavior "after" β (right).

p from $N_d(\beta)$ corresponds to removing $b_1, \ldots, b_k$ from β, and if all conditions $b_1, \ldots, b_k$ are implicit, then the behavior of β is preserved. In other words, we may consider p as implicit w.r.t. β.

Definition 7 (im1-implicit places)**.** Let β be a branching process, $\sim$ a folding equivalence, and $N = \beta_\sim$ the folded net. Let $B' \subseteq \mathit{imp}(\beta)$ be a consistent subset of implicit conditions of β. A place p of N is *im1-implicit* (w.r.t. B') iff $p \subseteq B'$.

By definition, removing all im1-implicit places from N simplifies the net and preserves the behavior of β (and hence of L). The places highlighted in the net of Fig. 12(top left) are classified as im1-implicit, but not as classically implicit; removing these places results in the net shown in Fig. 12(bottom left). We observed in experiments that this notion of implicit places identifies on average twice as many implicit places as the classical notion (Def. 5).

Figure 13 illustrates the difference between classically implicit places and im1-implicit places. Assume the net N and the log $L = [\mathsf{wxyz}, \mathsf{wz}]$ to be given. The L-induced unfolding β of N is shown in Fig. 13(middle). Folding β yields N again. Condition b_2 is implicit in β which makes c an im1-implicit place of N. Removing c corresponds to removing b_2: it preserves the behavior of N w.r.t. β, but it generalizes the behavior "after" β. The net without c has the occurrence sequence wxyx which cannot occur in N as y consumed the token on c needed by x. The branching process of N without c shown in Fig. 13(right) illustrates the situation.

5.2.2. Limit generalization

Removing im1-implicit places usually generalizes the behavior of N beyond β, i.e., behavior that is not described by the original log L. This generalization

can be limited by considering more behavior of N to be preserved, i.e., by extending β with more events of N to a branching process β^+ and then remove places from N based on implicit conditions of β^+. Experiments have shown that the branching process "after" β tends to grow very quickly for nets discovered by process discovery algorithms. We found the "single extension by all enabled events" to be feasible.

Definition 8 (im1$^+$-implicit places)**.** Let β be a branching process, $\sim$ a folding equivalence, and $N = \beta_\sim$ the folded net.

1. For each transition $t \in T_N$, let $en(t, \beta)$ be the set of all enabling locations of t in β (locations in β where t is enabled but did not occur, see Sect. 2.3). Let β^+ be the branching process obtained by extending β, for each $t \in T_N$ and each $B^* \in en(t, \beta)$ with a fresh t-labeled event e^*, ${}^\bullet e^* = B^*$.
2. Let $B' \subseteq imp(\beta^+)$ be a consistent subset of implicit conditions of β.
3. A place p of N is *im1$^+$-implicit* iff $p \subseteq B'$.

Each im1$^+$-implicit place is also an im1-implicit place, but not vice versa. An additional event e in β^+ turns an im1-implicit condition $b \in {}^\bullet e$ into a non-implicit condition if b is really needed to enable e, e.g., if ${}^\bullet e = \{b\}$. Also, there are im1$^+$-implicit places that are not classically implicit.

Figure 12(top right) shows the net from top left after removing im1$^+$-implicit places, the remaining im1-implicit places are highlighted red. We could confirm that removing im1$^+$-implicit places generalizes the behavior of N less than removing im1-implicit places. Yet, removing im1-implicit or im1+-implicit places yields almost the same reduction in terms of model complexity.

5.2.3. Stronger simplification

The two previous notions require all conditions that constitute a place to be implicit. Alternatively, we may classify a place as implicit as soon as *some* of its constituting conditions is implicit.

Definition 9 (im2-implicit places)**.** Let β be a branching process, $\sim$ a folding equivalence, and $N = \beta_\sim$ the folded net. Let $B' \subseteq imp(\beta)$ be a consistent subset of implicit conditions of β.

1. A place p of N is *im2-implicit* iff $p \cap B' \neq \emptyset$.
2. A place p of N is *im2$^-$-implicit* iff $p \cap imp(\beta) \neq \emptyset$.

Im2-implicit places are much more liberal than im1-implicit places because a single implicit condition suffices to characterize the place as implicit. The idea for this notion is to superimpose the fact that p is implicit in some situations to all situations of p. Im2$^-$-places are even more general as the implicit condition does not even have to be from a consistent subset of implicit conditions.

Removing all im2-implicit (im2$^-$-implicit) places from N could yield transitions without pre-place (which then can occur unboundedly) or transitions without post-place (which then have no effect on the net). To preserve a minimum of discovered process logic in N, we remove im2-implicit (im2$^-$-implicit) places from N 1-by-1 as follows.

- Let $\langle p_1, \ldots, p_k \rangle$ be an enumeration of the im2-implicit places (im2$^-$-implicit) of N.
- For $i = 1, \ldots, k$, if p_i is the only pre-place of its post-transitions or the only post-place of its pre-transitions, then keep p_i in N, otherwise remove p_i from N.

Applying this procedures usually generalizes behavior in β (in contrast to removing im1-implicit places). However, we observed in experiments significant structural simplifications that outweigh generalization by large. In some cases, the structure simplified by up to 72%; up to 95% of the places were implicit. Figure 12(bottom right) shows the net from the top left after removing all im2-implicit places; also the bottom left net highlights the im2-implicit places that are not im1-implicit.

5.3. Abstracting substructures: chains of unrestricted transitions

The previously presented two operators, unfolding/refolding and removing implicit places, generalized and simplified N along the structure of N as it was defined by the discovery algorithm $\mathcal{M}$ that returned N. Next, we present two operators to generalize N by *changing* N's structure.

Petri nets discovered through process discovery often contain several unrestricted transitions which are always enabled such as transition y in Fig. 5. The branching process then contains a chain of occurrences of these transitions that often cannot be folded to a more implicit structure as illustrated by e_2 and e_5 of Fig. 10.

Yet, we can abstract such a chain $t_1 \ldots t_n$ of unrestricted transitions with the same label a to a loop of length 1: (1) replace $t_1 \ldots t_n$ with a new transition t^* labeled a, (2) add a new place p^* in the pre- and post-set of t^*, and (3) for each place p which had a t_i in its pre-set and no other transition $t_j \neq t_i$ in its post-set, add an arc (t^*, p). Fig. 10 illustrates the abstraction: abstracting the chain of y-labeled transition of $N_i(\beta)$ (middle) yields the net $N_c(\beta)$ (right); we observed significant effects of this abstraction in industrial case studies.

The new transition t^* can mimic the chain $t_1 \ldots t_n$: t^* is always enabled and an occurrence of t^* has the combined effect of all $t_1, \ldots, t_n$. For this reason, a chain-abstracted net exhibits at least the behavior of the original net and possibly more. For longer chains this results in a significant reduction in size.

5.4. Splitting flower places

The last operation in this paper deals with a specific kind of places that are introduced by some discovery algorithms and cannot be abstracted with the previous techniques. We deal with these places by *splitting* them into several places.

A *flower place* p is place which has many transitions that contain p in their pre- and their post-set. Mostly, p only sequentializes occurrences of these transitions as can be seen in Fig. 14 to the left. Here, f can be viewed as a flower

Figure 14: The flower place f in the net on the left sequentializes occurrences of w and z. Splitting f and removing self-loops yields the structurally simpler net on the left with more behavior.

place. For example, z may occur arbitrarily often before or after w, though only after x and before y occurred. Similarly, w can only occur after x and before y. However, as w can occur only once, the restrictive effect of flower place f on w is limited.

Based on this observation we may (1) remove self-loops of transitions that are still restricted by another pre-place such as w, and (2) split the flower place for a transition t that has no other pre-place, i.e., to create a new place p in the pre- and post-set of t. The net in Fig. 14 to the right shows the result of this abstraction. The resulting net exhibits more behavior than the original net. Some of this behavior may even be not explained by the log. For example, w may occur now before x and after y. Yet, the transformation may help to significantly reduce the number of synchronizing arcs in the discovered process model. We observed a removal of up to 95% of the arcs leading to structural simplification of the same amount.

5.5. A Parameterized Process Model Simplification Algorithm

All operations presented in the preceding sections together yield the following algorithm for simplifying a Petri net $N = \mathcal{M}(L)$ that was discovered from a log L by a process discovery algorithm $\mathcal{M}$ (see also Fig. 4).

1. Construct the branching process $\beta(L)$ of N that represents all cases $l \in L$; optionally filter infrequent behavior from $\beta(L)$ (see Sect. 3.3); construct the folding equivalence $\sim = det(future(\beta(L)))$; fold $N_d := \beta(L)_\sim$.
2. Remove implicit places from N_d (using one of the notions presented in Sect. 5.1 and 5.2).
3. Abstract chains of unrestricted actions from N_d.
4. Split flower-places of N_d.

The technique is *modular*. By design, the result of each transformation step is a net that is structurally simpler than the preceding net and can replay the entire log L, i.e., the resulting model N' *recalls* each case of L. Moreover, starting from $\beta(L)$ which is an overfitting model of L, each step also *generalizes* $\beta(L)$ towards an underfitting model of L. The degree to which N' allows more behavior than L is measured by a *precision* metric [26, 27]. Each of the four steps can be applied

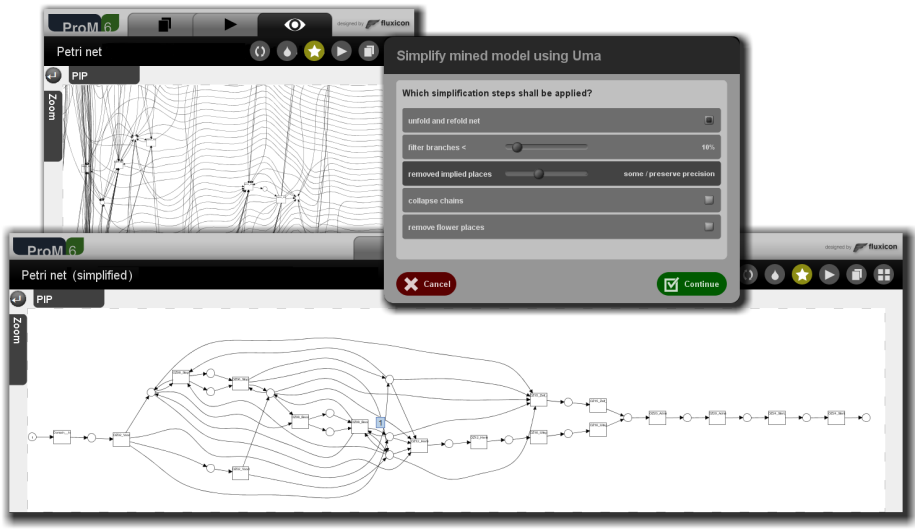

Figure 15: The configuration screen of the ProM plugin "Uma" and an example of a simplified model.

selectively. This way it is possible to balance between precision, generalization, and complexity reduction.

The algorithm described above assumes a perfectly aligned event log where all behavior is considered to be relevant. In Sect. 3.1, we showed how to align log and model by massaging the event log. Alignment is considered to be a preprocessing step conducted before executing the above algorithm.

6. Experimental Results

This section describes the implementation of the techniques introduced in the preceding sections and reports on experimental results. All result were obtained using our implementation in ProM.

6.1. Implementation in ProM

We implemented our approach as a plugin for the process mining toolkit ProM. ProM as well as the package "Uma" that contains the plugin are available at `http://www.processmining.org/`.

To use the plugin called "simplify mined models", the user picks as input a log and a Petri net that was discovered from this log. The plugin then shows a panel (Fig. 15), where the user can configure the simplification of the net by disabling any of the steps as discussed in Sect. 5.5 and by choosing the amount of simplification and generalization to be applied when removing implicit places. By default, all steps are enabled requiring no Petri net knowledge from the user for model simplification; the default setting for implicit places is "im2" (Def. 9).

The plugin then runs fully automatically and returns the simplified Petri net which can be inspected and analyzed further as shown in Fig. 15.

Our implementation applies a number of optimizations to increase performance. The log-induced unfolding is built using data structures similar to the ones used for constructing classical unfoldings of Petri nets [22]. The unfolding is only refolded after the entire folding relation has been constructed; this way we only have to handle equivalent sets of nodes rather than change the structure of a Petri net. Because the unfolding can become very large, we save memory by detecting implicitness of a condition by depth-first search on the unfolding's structure (see Sect. 5.2); intermediate results are cached to improve search performance. A canonical order of nodes in the unfolding allows to truncate parts of the search space that contain no predecessor of a node. Our current implementation does not search for a largest consistent set of implicit conditions (Def. 6) and picks conditions in a greedy way until no other condition is implicit; future improvements are possible. We use an extension library of the Vip-Tool [28] to identify classically implicit places using an ILP solver.

6.2. Effect and Comparison of Simplification Steps

Using this plugin, we validated our approach in a series of experiments on benchmark logs, and logs obtained in industrial case studies. For each experiment, we generated from a given log L a Petri net N with the ILP Miner [9] using its default settings; the log was *not* pre-processed beforehand. We then applied the simplification algorithm of Sect. 5.5 on N using the original log L. Figs. 1 and 2 illustrate the effect of our algorithm on industrial processes: the algorithm balances the control-flow logic of a discovered process model by removing up to 85% of the places, and up to 92% of the arcs.

6.2.1. Effect of Complete Simplification Procedure

Table 1 gives some more details when applying all reduction steps and removing im2-implicit places (which we consider the most suitable setting for model simplification). The logs named aXnY are benchmark logs having X different activities; the aXn0 are logs of highly structured processes. Y is the percentage of random noise events introduced into the log aXn0. The remaining logs were obtained in case studies in the health care domain (HC) and from municipal administrations (M). We compared the nets in terms of the numbers $|P|$, $|T|$, and $|F|$ of places, transition and arcs, and their simple graph complexity $c = \frac{|F|}{|P|+|T|}$ which roughly correlates with the perceived complexity of the net. The effect of the algorithm is measured as the percentage of places $|P|$ and arcs $|F|$ removed from (or added to) the original net, and the percentage by which the graph complexity was reduced.

The numbers show that almost all models could be reduced significantly in terms of places and arcs (up to 85% and 92% with 53% and 67% in average). We observed that some models (a32n0, M1) grew slightly in size, i.e., more places and transitions were introduced. We found unrolled loops of length greater than 2 that occur only once in the branching process to be responsible for the

Table 1: Experimental results using all reduction steps and removing im2-implicit places.

	original $\lvert P\rvert$/ $\lvert T\rvert$/ $\lvert F\rvert$/ c	**simplified** $\lvert P\rvert$/ $\lvert T\rvert$/ $\lvert F\rvert$/ c	**difference** $\lvert P\rvert$/ $\lvert F\rvert$/ c	**runtime** in sec
a22n00	21/ 22/ 60/ 1.40	20/ 22/ 54/ 1.29	-5%/ -10%/ -8%	0.819
a22n05	38/ 22/204/ 3.40	19/ 22/ 77/ 1.89	-50%/ -62%/ -45%	2.574
a22n10	52/ 22/428/ 5.78	14/ 22/ 78/ 2.17	-73%/ -82%/ -63%	29.2
a22n20	74/ 22/569/ 5.93	14/ 22/ 58/ 1.61	-81%/ -90%/ -73%	131.9
a22n50	91/ 22/684/ 6.05	15/ 22/ 52/ 1.41	-84%/ -92%/ -77%	163.5
a32n00	32/ 32/75/ 1.17	32/ 32/ 74/ 1.16	0%/ -1%/ -1%	0.243
a32n05	44/ 32/225/ 2.96	34/ 32/ 107/ 1.62	-23%/ -52%/ -45%	5.1
a32n10	68/ 32/543/ 5.43	24/ 32/ 111/ 1.98	-65%/ -80%/ -63%	83.6
a32n20	90/ 32/612/ 5.02	28/ 32/ 98/ 1.63	-69%/ -84%/ -67%	108.8
a32n50	110/ 32/868/ 6.11	27/ 32/ 102/ 1.73	-75%/ -88%/ -72%	173.4
HC1	41/ 15/224/ 4.00	10/ 15/ 44/ 1.76	-76%/ -80%/ -56%	0.029
HC2	20/ 14/139/ 4.09	8/ 14/ 48/ 2.18	-60%/ -65%/ -47%	0.003
HC3	23/ 14/122/ 3.30	11/ 14/ 42/ 1.68	-52%/ -66%/ -49%	0.003
HC4	43/ 17/224/ 3.73	11/ 17/ 47/ 1.69	-74%/ -79%/ -55%	0.028
HC5	89/ 26/959/ 8.34	13/ 26/ 79/ 2.03	-85%/ -92%/ -76%	0.362
M1	58/ 55/358/ 3.17	64/ 81/ 196/ 1.35	10%/ -45%/ -57%	0.163
M2	31/ 23/255/ 4.72	17/ 23/ 64/ 1.60	-45%/ -75%/ -66%	0.038
avg	54/ 26/ 385/ 4.4	21/ 27/ 78/ 1.7	-53%/ -67%/ -54%	41.2
max			-85%/ -92%/ -77%	

growth in size. Our algorithm cannot fold back these singular loops; though the algorithm could be extended to handle such patterns. Yet, in all cases where the logs contained noisy behavior, our technique reduced each Petri net's graph complexity c by 45% to 77% (with 54% in average). A modeler is able to inspect models of this complexity and gain an understanding of the modeled process as illustrated by Figs. 1 and 2 which show the models of HC1 and M2.

6.2.2. Effect of Individual Simplification Steps

We also investigated the effect of the individual simplification steps and compared the different notions of implicit places introduced in Section 5.2. Figure 16 shows how model complexity changed through the different stages of the algorithm in comparison to the notion of implicit places that was removed in the corresponding step; the diagrams show minimum, average, and maximum graph complexity in our data set.

It turned out that unfolding/refolding in most cases only yields a reduction of up to 5%, with one exception from the industrial logs yielding about 30% reduction. The different modes of removing implicit places have different effects:

1. Removing classical implicit places (im) reduces complexity by at most 5%. In average, 3 places were removed from the model.
2. Removing im1- and im1$^+$-implicit places reduces complexity by about 5%. About 6 places were removed by im1$^+$ and 6.5 places by im1 in average.
3. Removing im2- and im2$^-$-implicit places yields significant reduction for all logs that contain noise, ranging from 40% up to 60% reduction in complexity. In average, 50% of the places (23 places) were removed.

Chain reduction removed between 12% and 51% of the transitions in the refolded

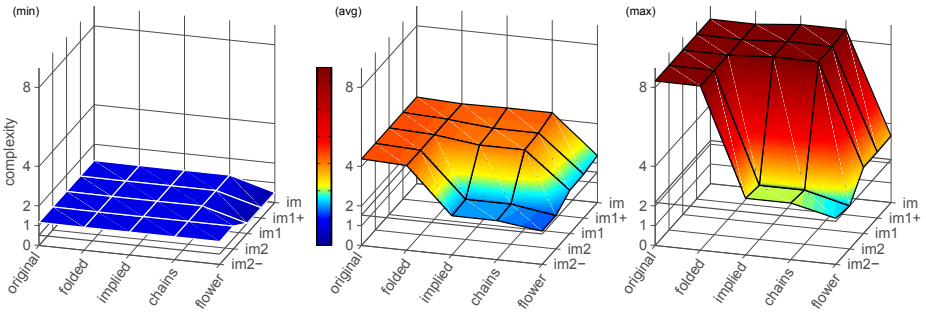

Figure 16: Accumulated effect of model simplification depending on the kind of implicit places being reduced.

model of logs with 5% noise and in M1; as this reduction removes transitions, places, and arcs, it has no measurable effect in the graph complexity.

From models obtained by removing classically/im1/im1$^+$-implicit places, splitting flower places allows to remove up to 95% of the arcs. In particular, models with a high level of noise tend to have a large number of arcs connected to flower places. After removing im2- and im2$^-$-implicit places, splitting flower places can remove about 10% of the arcs in logs with a noise level of 10% or more, other arcs already had been removed beforehand by removing their adjacent implicit places.

Runtimes correlate with the size of the branching processes constructed by the algorithm, we observed branching processes of up to 192,000 nodes and 360,000 arcs in the benchmarks and 4,800 nodes and 9,800 arcs in the case studies. Runtime is dominated by computing the folding equivalence: roughly 2/3 of the runtimes shown in Tab. 1 are used for folding. The runtimes for unfolding, chain abstractions, and splitting flower are neglible. Runtimes for removing implicit places depend on the notion:

1. Classically implicit places can be removed in at most 3 sec (1sec in avg.).
2. Removing im1-implicit places succeeds in milliseconds.
3. Removing im1$^+$-implicit places takes significantly more time as the unfolding needs to be extended first (about 50% of the time is required for folding, i.e., max. 30 secs, avg. 3 secs).
4. Removing im2-implicit places takes runtimes up to 54 secs, 9 secs in average. Here we have to first find a consistent set of implicit conditions before removing places. For im1- and im1$^+$-implicit places this set can be built much faster by exploiting that implicit conditions of interest are folding equivalent.
5. im2$^-$-implicit places avoid computing a consistent set and hence computation finishes in less than a second.

Compared to an earlier version of the paper [29] optimizations in the implementation (see Sect. 6.1) reduced runtime by about factor 6.

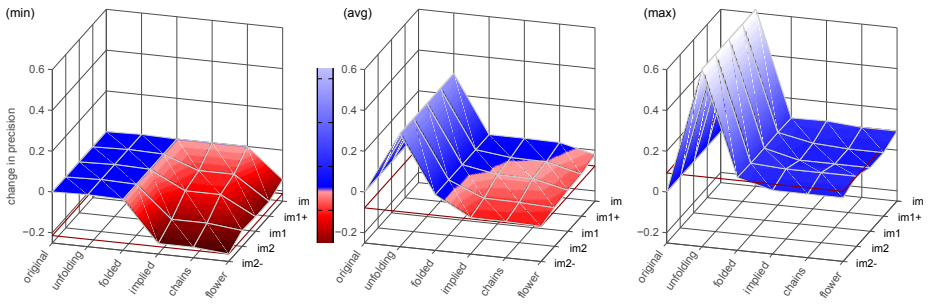

Figure 17: Change in precision depending on the kind of implicit places being reduced.

6.2.3. Generalization of Behavior

Finally, we measured the amount of behavioral generalization that is introduced by the different simplification steps. In particular, we compared the effect of the different notions of implicit places. We measured generalization by comparing how much more behavior a model allows compared to its original log, this measure is known as *precision*. In our experiment, we applied the measure proposed in [27, 30]. A precision value of 1.0 means the model allows for exactly the behavior in the model; the lower the precision, the more behavior does the model allow.

In the diagram in Fig. 17 the x-axis shows the different stages of the simplification algorithm (including the log-induced unfolding), the y-axis the different notions of implicit places, and the z-axis the *relative* change of precision compared to the precision of the original model, for the worst case, the average case, and the best case.

As expected, we could observe a significant increase in precision for the log-induced unfolding (reaching $+0.5$ in average and up to $+0.61$ in the best case). The unfolding presents just the behavior of the log and additional linearizations of concurrent transitions that are not represented in the log (thus, an unfolding usually does not have perfect precision of 1.0). Refolding the unfolding increased precision by up to 0.1 in the best case and just slightly in average, compared to the original model. When removing implicit places, precision drops. The amount of generalization (i.e., loss in precision) depends on the notion of implicit places.

For the classical notion (Def. 5), precision was preserved in all cases. Removing only im1$^+$-implicit places still yielded increased precision in the average case and a loss of about -0.05 in the worst case. The more liberal the notion of implied places, the more precision is lost, with -0.15 (-0.19) in the worst case for im2- (im2$^-$)-implicit places in the worst case.

When applying alls steps, the effect for different notions of implicit places converge, ranging from -0.14 precision for classically implicit places to -0.21 for im2$^-$-implicit places in the worst case. Most importantly, in average we observed only a very small loss of precision by -0.02 to -0.07 whereas precision did increase for some cases between $+0.06$ and $+0.09$.

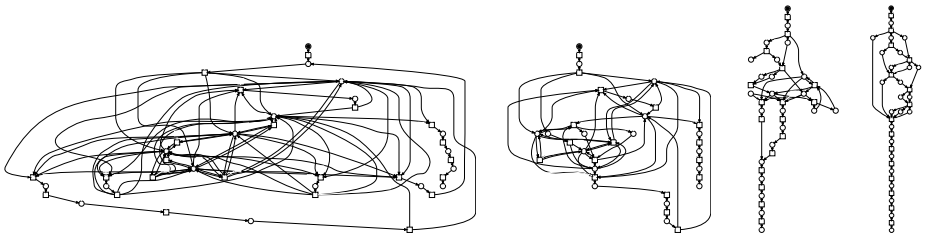

Figure 18: Filtering the model of Fig. 2 with thresholds $f = 0, 0.05, 0.15, 0.3$, and removing im1$^+$-implicit places.

6.3. Effect of Filtering Infrequent Behavior

We just presented the effects of the simplification algorithm that preserved all behavior of the log. Our approach also allows to remove infrequent behavior from the branching process as described in Sect. 3.3 in order to remove behavior that requires complex model structures. To measure the effect of the filtering, we implemented the following relative filter: for a filter threshold f, remove from the branching process every event e (and its successors) where $\kappa(e)/\kappa(b) < f$, where $b \in {}^\bullet e$ is the pre-condition of e with the smallest κ-value. The rationale for this filter is that the least-frequent pre-condition of e determines how often e could have occurred at most.

In the experiment, models were unfolded, filtered, folded, and finally im1$^+$-implicit places were removed. We measured the effect of this filter for values $f = 0.05, 0.1, 0.15, 0.2, 0.3$.

Figure 18 illustrates the effect of the filtering on the model of Fig. 2; we depict the unfiltered model $f = 0$ and the models for $f = 0.05, 0.15, 0.3$. The filtering of infrequent behavior (noise) immediately results in less complex models. For $f = 0.05$ some structured process logic is recognizable in the model; for $f = 0.15$ the model is rather structured with some complex logic in the middle; $f = 0.30$ filters all behavior but the most frequent path, the model has no alternative executions.

Figure 19 shows how model complexity changes in the entire data set through the different stages of the simplification algorithm. We depict average (left) and maximal model complexities (right) for filter values $f = 0.05$ (top) and $f = 0.30$ (bottom); the results for the other filter values are linearly distributed between these two extremes. For models that already were structurally simple, no difference could be noted between the unfiltered and the filtered cases.

Model complexity falls as more behavior is filtered from the branching process ($f = 0.05$ vs $f = 0.30$). Filtering is particularly effective on models discovered from logs with much infrequent behavior (e.g., benchmark logs with noise $\geq 10\%$). The main effect of filtering can be seen before flower places are split. After flower places are split, the complexities converge in all cases to similar values. This suggests that all behavior (including the frequent behavior) requires a structure of certain complexity that cannot be simplified *without* generalizing the model's behavior.

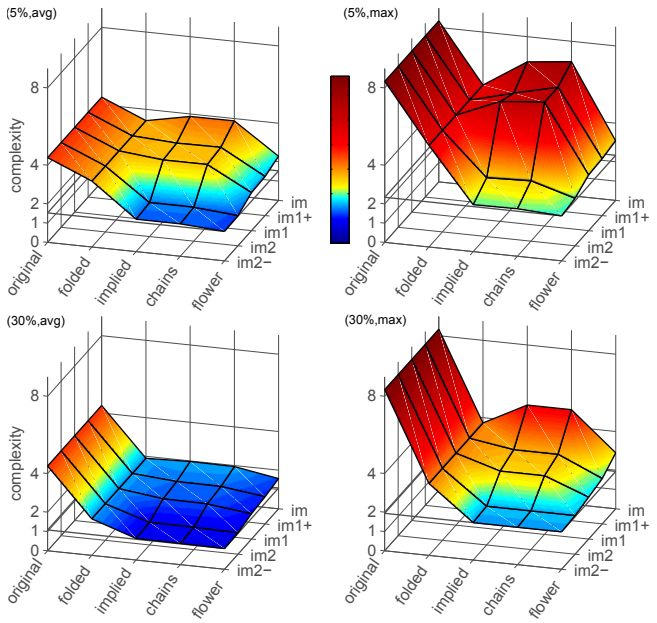

Figure 19: Change in complexity by filtering infrequent behavior.

When comparing simplification by removing implicit places with simplification by filtering, the resulting models differ significantly depending on how the structure is simplified, i.e., compare Fig. 18(filtering) and Fig. 2(right, removing implicit places and flower places). We observed growing model complexity for some models when implicit places were removed after filtering, see Fig. 19(right). This is mostly due to places with few incoming or outgoing arcs that are obtained by filtering (low complexity) and that are then classified as implicit places afterwards and removed (rising complexity).

Filtering removes infrequent behavior from the branching process. The filtered behavior is (most likely) no longer a behavior of the refolded model, i.e., the model cannot replay the complete original log but only its frequent behavior. To measure the deviation introduced by filtering, we also measured the fitness of each filtered, simplified model to its original log using the technique of [17, 18]. Figure 20 shows how fitness of the resulting model changed depending on the filter threshold and the notion of implicit places; the least fitness (left) and the average fitness (right) of the data set are shown; a model that can replay all traces has fitness 1.0.

In all models discovered from logs with noise, filtering results in models that cannot replay the entire log anymore (falling fitness). For low filtering thresholds ($< .1$) log and model deviate only in 1 or 2 events per non-fitting cases which still leads to high fitness values of approx .95. The more behavior is filtered, the more cases cannot be replayed on the model. We observed a steep decline in fitness for filter thresholds of $f = 0.3$ and more. Here also the frequent behavior

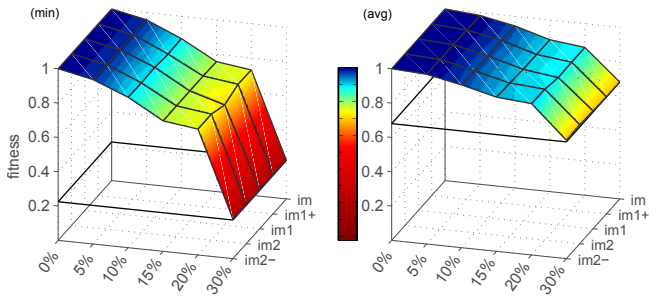

Figure 20: Change in fitness depending on thresholds for filtering infrequent behavior.

gets filtered. The notion of implicit places has no significant impact on fitness.

Throughout the experiment, most effective results were achieved for logs with many cases. In these logs, frequent and infrequent behavior can clearly be distinguished which leads to effective filtering as demonstrated in Fig. 18. For logs that consist of only few or many unique cases, we observed that all behavior was filtered for filter thresholds > 0.05 (as the branching process contained no frequent behavior). For these logs, filtering is inapplicable.

6.4. Discussion

In sight of these observations, we can conclude that our proposed technique for simplifying discovered process models succeeds. Typically, the structure of a discovered model can be simplified dramatically while *fitness is preserved completely*.

At the same time, we could confirm that this simplification comes with a reasonable model generalization w.r.t. the behavior recorded in the log. In average, model behavior generalizes only slightly, though in some cases the model can be generalized considerably, depending on the chosen parameters. The choice in the notion of implicit places allows to trade simplification vs. generalization (i.e., im1 vs. im2) as well as generalization vs. runtime (i.e., im1^+ vs. im1 and im2 vs. im2^-).

The model can also be simplified by removing infrequent behavior through filtering. This allows to trade simplification for specialization: the filtered model is significantly simpler but only replays a subset of the log. We also observed filtered models to be more structured than unfiltered models (i.e., compare Fig. 2 and Fig. 18).

Altogether, we found a new way to let a user *balance* between the 4 competing quality criteria of process models: fitness, simplicity, generalization (not overfitting) and precision (not underfitting) [1]. In particular, our technique allows to improve one criterion (simplicity) while balancing the other criteria in a controlled way. However, our approach should not be seen as a black box: a user still has to choose which criteria she wants to optimize for, e.g., simplicity vs. precision.

7. Related work

In the last decade, process mining emerged as a new research disciple combining techniques from data mining, process modeling, and process analysis. Process discovery, i.e., constructing a process model based on sample behavior, is the most challenging process mining task [1, 3] and many algorithms have been proposed [2]. Examples are the α algorithm [13], variants of the α algorithm [14, 15], heuristic mining [4], genetic process mining [6, 7], fuzzy mining [5], process mining techniques based on language-based regions [8, 9], mining based using convex polyhedra [10], and various multi-phase approaches [11, 12].

The approach presented in this paper can be used to simplify the models generated by all of the approaches mentioned above. However, there are two relevant characteristics. The first characteristic is whether the discovery algorithm produces a Petri net or a process model in some other notation (e.g., heuristic nets, C-nets, EPCs, or plain transition systems). The techniques described in [8–10, 13–15] produce directly a Petri net. For the other approaches some conversion is needed [7, 16]. The second characteristic is whether the discovery algorithm guarantees a fitness of 1. Most techniques do not provide such a guarantee. Notable exceptions are [8–12]. However, these techniques cannot deal with noise. Therefore, we showed that it is possible to align log and model in Sect. 3.1. This way we can also handle process models that are not perfectly fitting. For aligning traces without an initial model we refer to [31].

The approach of [12] allows to balance between overfitting and underfitting of mined process models, controlled by the user. However, this approach requires expert knowledge to find the right balance. Our approach is easier to configure, and yields significant simplification in the fully automatic setting. Moreover, our approach even simplifies models produced by [12] as illustrated by Fig. 3.

Our approach can improve results of existing discovery algorithms because all reasoning on how to improve a model is done on the *log-induced unfolding*. This unfolding provides a *global view on behavior* that is directly related to the model's *structural features* and contains information about *frequencies* and *redundancies* of model elements w.r.t. the given log. Such combined information is *not available* to most process discovery algorithms during the discovery.

Conformance checking techniques [17, 18, 26, 27, 30], like the post-processing approach presented in this paper, use a log and a model as input. In [26], the log is replayed on the Petri net to see where the model and reality diverge. In [27, 30], the behavior of the model restricted to the log is computed. The border between the log's and model's behavior highlights the points where the model deviates from the log. In [17, 18], the problem of aligning model and log is translated into an optimization problem to find the "closest" path in the model.

The goal of this paper is to simplify and structure discovered process models. This is related to techniques discovering block structures in graph-based models [32, 33] and transforming unstructured models into structured ones [34]. These techniques focus on and preserve concurrency in models as in our approach. In particular the future equivalence used in this paper to fold the log-based unfolding preserves *fully concurrent bisimulation* [35] that is also used

in [34]. However, these techniques just consider the model as is and not the behavior that has been observed in reality and recorded in a log. Moreover, [34] focuses on equivalent transformations whereas we allow to generalize or to filter behavior to trade competing quality criteria if needed.

The problem coped with in this paper resembles the problem of restricting a system (here N) to admissible behaviors (here L) by means of a *controller*, e.g., [36]. However, these approaches require N to have a finite state space, which usually does not hold for discovered process models. Additionally, our aim is also to structurally simplify N, not only to restrict it to L.

This paper is an extended version of our BPM 2011 paper [29]. Compared to [29] we extended our approach to be able to deal with an initial model that does not fit completely. Moreover, we showed that the overfitting branching process can be extended with frequencies to support the removal of noise and outliers. We also improved the identification of implicit places using a notion tailored towards unfoldings. For example, we showed how different notions of implicit place allow to trade model simplification and generalization. In addition, we presented various new experimental results that also document an acceptable generalization by our technique despite significant model simplification. These results also show that we were able to further improve the approach already presented in [29].

8. Conclusion

The approach presented in this paper can be combined with any process discovery technique that produces a Petri net that can reproduce the event log. Extensive experimentation using real-life event logs show that our post-processing approach is able to dramatically simplify the resulting models. Moreover, the approach allows users to balance between overfitting and underfitting. Unnecessary generalization is avoided and the user can guide the simplification/generalization process.

Our decisive technical contribution to process mining is the adaptation of classical branching processes to *log-induced unfoldings*. The latter provides a global view on the behavior in the log that is related to the model structure and contains information about frequencies, redundancies, and equivalences. This allows to to balance quality criteria for process models in a different way than before.

Acknowledgements. The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 257593 (ACSI).

References

[1] W. M. P. van der Aalst, Process Mining: Discovery, Conformance and Enhancement of Business Processes, Springer, 2011.

[2] B. F. van Dongen, A. K. Alves de Medeiros, L. Wen, Process Mining: Overview and Outlook of Petri Net Discovery Algorithms, ToPNOC 2 (2009) 225–242.

[3] IEEE Task Force on Process Mining, Process Mining Manifesto, in: BPM Workshops, Vol. 99 of LNBIP, Springer, 2011.

[4] A. J. M. M. Weijters, W. M. P. van der Aalst, Rediscovering Workflow Models from Event-Based Data using Little Thumb, Integrated Computer-Aided Engineering 10 (2) (2003) 151–162.

[5] C. W. Günther, W. M. P. van der Aalst, Fuzzy Mining: Adaptive Process Simplification Based on Multi-perspective Metrics, in: BPM 2007, Vol. 4714 of LNCS, Springer, 2007, pp. 328–343.

[6] A. K. Alves de Medeiros, A. J. M. M. Weijters, W. M. P. van der Aalst, Genetic Process Mining: An Experimental Evaluation, Data Mining and Knowledge Discovery 14 (2) (2007) 245–304.

[7] W. M. P. van der Aalst, A. K. Alves de Medeiros, A. J. M. M. Weijters, Genetic Process Mining, in: G. Ciardo, P. Darondeau (Eds.), Applications and Theory of Petri Nets 2005, Vol. 3536 of LNCS, Springer, 2005, pp. 48–69.

[8] R. Bergenthum, J. Desel, R. Lorenz, S. Mauser, Process Mining Based on Regions of Languages, in: BPM 2007, Vol. 4714 of LNCS, Springer, 2007, pp. 375–383.

[9] J. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, A. Serebrenik, Process Discovery using Integer Linear Programming, Fundamenta Informaticae 94 (2010) 387–412.

[10] J. Carmona, J. Cortadella, Process Mining Meets Abstract Interpretation, in: J. Balcazar (Ed.), ECML/PKDD 210, Vol. 6321 of Lecture Notes in Artificial Intelligence, Springer, 2010, pp. 184–199.

[11] B. F. van Dongen, W. M. P. van der Aalst, Multi-Phase Process Mining: Building Instance Graphs, in: ER 2004, Vol. 3288 of LNCS, Springer, 2004, pp. 362–376.

[12] W. M. P. van der Aalst, V. Rubin, H. M. W. Verbeek, B. F. van Dongen, E. Kindler, C. W. Günther, Process mining: a two-step approach to balance between underfitting and overfitting, Software and System Modeling 9 (1) (2010) 87–111.

[13] W. M. P. van der Aalst, A. J. M. M. Weijters, L. Maruster, Workflow Mining: Discovering Process Models from Event Logs, IEEE Transactions on Knowledge and Data Engineering 16 (9) (2004) 1128–1142.

[14] L. Wen, W. M. P. van der Aalst, J. Wang, J. Sun, Mining Process Models with Non-Free-Choice Constructs, Data Mining and Knowledge Discovery 15 (2) (2007) 145–180.

[15] L. Wen, J. Wang, W. M. P. van der Aalst, B. Huang, J. Sun, Mining Process Models with Prime Invisible Tasks, Data and Knowledge Engineering 69 (10) (2010) 999–1021.

[16] W. M. P. van der Aalst, A. Adriansyah, B. F. van Dongen, Causal Nets: A Modeling Language Tailored Towards Process Discovery, in: J. Katoen, B. Koenig (Eds.), 22nd International Conference on Concurrency Theory (CONCUR 2011), LNCS, Springer, 2011, pp. 28–42.

[17] A. Adriansyah, B. F. van Dongen, W. M. P. van der Aalst, Towards Robust Conformance Checking, in: M. Muehlen, J. Su (Eds.), BPM 2010 Workshops, Proceedings of the Sixth Workshop on Business Process Intelligence (BPI2010), Vol. 66 of LNBIP, Springer, 2011, pp. 122–133.

[18] A. Adriansyah, B. van Dongen, W. M. P. van der Aalst, Conformance Checking using Cost-Based Fitness Analysis, in: IEEE International Enterprise Computing Conference (EDOC 2011), IEEE Computer Society, 2011.

[19] U. Goltz, W. Reisig, Processes of Place/Transition-Nets, in: ICALP'83, Vol. 154 of Lecture Notes in Computer Science, Springer, 1983, pp. 264–277.

[20] J. Engelfriet, Branching Processes of Petri Nets, Acta Informatica 28 (6) (1991) 575–591. doi:http://dx.doi.org/10.1007/BF01463946.

[21] M. Nielsen, G. D. Plotkin, G. Winskel, Petri Nets, Event Structures and Domains, Part I, Theor. Comput. Sci. 13 (1981) 85–108.

[22] J. Esparza, S. Römer, W. Vogler, An Improvement of McMillan's Unfolding Algorithm, Formal Methods in System Design 20 (3) (2002) 285–310.

[23] J. Esparza, K. Heljanko, Unfoldings: A Partial-Order Approach to Model Checking, Springer, 2008. doi:10.1007/978-3-540-77426-6.

[24] J. Colom, M. Silva, Improving the Linearly Based Characterization of P/T Nets, in: Advances in Petri Nets 1990, Vol. 483 of LNCS, Springer, 1991, pp. 113–145.

[25] D. Fahland, From Scenarios To Components, Ph.D. thesis, Humboldt-Universität zu Berlin and Technische Universiteit Eindhoven (2010).

[26] A. Rozinat, W. M. P. van der Aalst, Conformance Checking of Processes Based on Monitoring Real Behavior, Information Systems 33 (1) (2008) 64–95.

[27] J. Muñoz-Gama, J. Carmona, A Fresh Look at Precision in Process Conformance, in: BPM'10, Vol. 6336 of LNCS, Springer, 2010, pp. 211–226.

[28] R. Bergenthum, J. Desel, S. Mauser, Comparison of different algorithms to synthesize a petri net from a partial language, in: Transactions on Petri Nets and Other Models of Concurrency III, Vol. 5800 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2009, pp. 216–243, 10.1007/978-3-642-04856-2_9.

[29] D. Fahland, W. M. P. van der Aalst, Simplifying Mined Process Models: An Approach Based on Unfoldings, in: S. Rinderle, F. Toumani, K. Wolf (Eds.), Business Process Management (BPM 2011), Vol. 6896 of LNCS, Springer, 2011, pp. 362–378.

[30] J. Muñoz-Gama, J. Carmona, Enhancing Precision in Process Conformance: Stability, Confidence and Severity, in: N. Chawla, I. King, A. Sperduti (Eds.), IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2011), IEEE, Paris, France, 2011.

[31] R. P. J. C. Bose, W. M. P. van der Aalst, Trace Alignment in Process Mining: Opportunities for Process Diagnostics, in: R. Hull, J. Mendling, S. Tai (Eds.), Business Process Management (BPM 2010), Vol. 6336 of LNCS, Springer, 2010, pp. 227–242.

[32] J. Vanhatalo, H. Völzer, J. Koehler, The Refined Process Structure Tree, Data Knowledge Engineering 68 (9) (2009) 793–818.

[33] A. Polyvyanyy, J. Vanhatalo, H. Völzer, Simplified Computation and Generalization of the Refined Process Structure Tree, in: WS-FM'10, Vol. 6551 of Lecture Notes in Computer Science, 2010, pp. 25–41.

[34] A. Polyvyanyy, L. García-Bañuelos, M. Dumas, Structuring acyclic process models, Inf. Syst. 37 (6) (2012) 518–538.

[35] E. Best, R. R. Devillers, A. Kiehn, L. Pomello, Concurrent Bisimulations in Petri Nets, Acta Inf. 28 (3) (1991) 231–264.

[36] A. Lüder, H.-M. Hanisch, Synthesis of Admissible Behavior of Petri Nets for Partial Order Specifications, in: WODES'00, Kluwer, 2000, pp. 409 – 431.