Simplifying Mined Process Models: An Approach Based on Unfoldings

Dirk Fahland and Wil M.P. van der Aalst

Eindhoven University of Technology, The Netherlands
{d.fahland,w.m.p.v.d.aalst}@tue.nl

Abstract. Process models discovered using process mining tend to be complex and have problems balancing between overfitting and underfitting. Overfitting models are not general enough while underfitting models allow for too much behavior. 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, e.g., 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 [1,2].

Input for process discovery is a collection of traces. Each trace describes the lifecycle 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. Moreover, information from the log can be projected on such models, e.g., showing bottlenecks and deviations.

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 tasks has 10! = 3628800 potential interleavings. However, event logs typically contain less 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. A model is underfitting when it overgeneralizes the example behavior in the log, i.e., the model allows for behaviors very
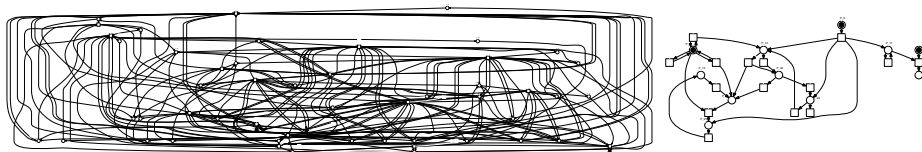
Fig. 1. Hospital patient treatment process after process mining (left) and after subsequent simplification using the approach presented in this paper (right)
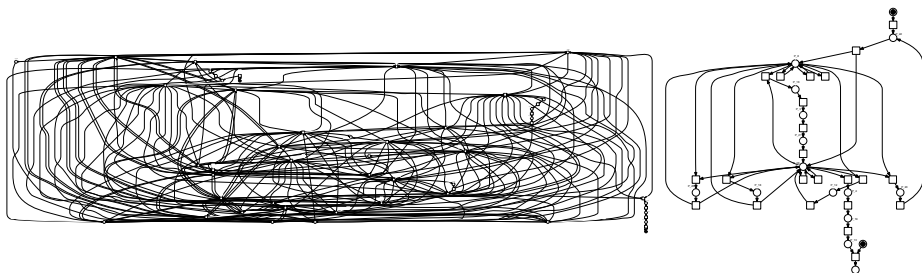

Fig. 2. Municipality complaint process after process mining (left) and after subsequent simplification (right)

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].

A variety of approaches has been proposed to address these challenges. Technically, all these approaches extract ordering constraints on tasks which are then expressed as control-flow constructs in the resulting process model. Usually, infrequent (exceptional) behavior in the log leads to complex control-flow constructs in the model. Different approaches cope with this problem in different ways.

(1) Heuristic mining [3] and fuzzy mining [4] abstract from infrequent behavior to simplify the model. Genetic algorithms [5] use evolution to find suitable models. As a downside, the resulting model *describes only an abstraction of the log*.

(2) Approaches that do not abstract from infrequent behavior tend to over-generalize to create a model that is able to replay the entire log. The approach presented in [6] guarantees that all traces in the log can be reproduced by the model. In [7] 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.

(3) Other approaches generalize by *restricting the most general model as much as possible*. Techniques based on *language-based regions* [8,9] use the property that adding a place in a Petri net restricts its behavior, i.e., a place can be seen as a constraint on the model's behavior. The Petri net with transitions T and without any places can reproduce any event log over T. As shown in [9] a system of inequations can be solved
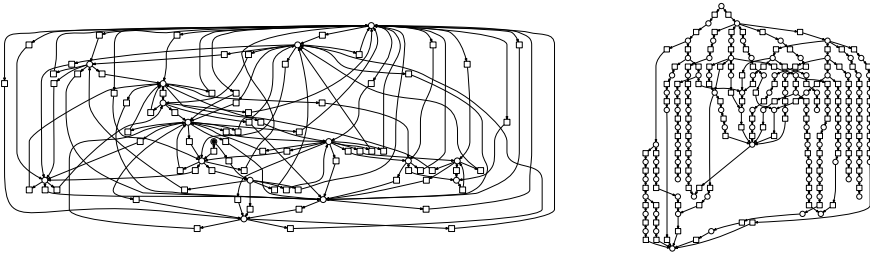
Fig. 3. The hospital process (Fig. 1) discovered by [10] (left) can be simplified (right)

to add places that do not exclude behavior present in the log. None of these approaches (2) and (3) allows the user to control the balance between overfitting and underfitting. Moreover, the resulting models tend to be convoluted as illustrated by Fig. 1 and 2.

(4) In [10] an approach to balance overfitting and underfitting is proposed. First, 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. But this approach requires expert knowledge to specify from the log transition system states that yield a balanced model. 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. In the following, we propose a technique for re-adjusting the generalization done by process mining algorithms, and to cut down involved process logic to the logic that is *necessary* to explain the observed behavior. Starting point for our approach is an event log L and a discovered process model $N = \mathcal{M}(L)$ where the mining algorithm $\mathcal{M}$ guarantees a fitting model, i.e., all traces in L can be replayed on N[1]. Next we generate a compact representation β of the behavior of the process model N w.r.t. the log L from which N was discovered. We then deliberate this representation β from unnecessary dependencies, and apply generalization techniques that do not introduce new dependencies unless motivated by a specific generalization aim.

Technically, we use Petri nets to represent process models; β is the branching process of the Petri net N representing the traces in L. We (1) define operations to fold β to a more explicit representation N' of the process logic compared to N, (2) reduce superfluous control-flow structures by removing implicit places from N', and (3) define abstraction operations to simplify the structure of N' and generalize the described behavior in a controlled way. Our technique leads to a *modular technique* for transforming N to a model N' that has a simpler structure than N. Moreover, if the original model N exhibits all behavior of L, then also the simplified model N' exhibits L. Fig. 4 illustrates

---

[1] If the mining algorithm does not guarantee a perfectly fitting model, we can still use the technique presented in [11] to create a fitting event log.
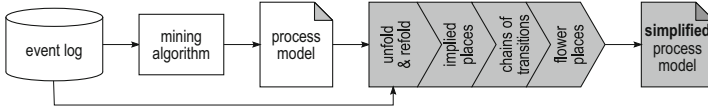
Fig. 4. Overview on the approach to simplify mined process models

our approach which is supported by the process mining toolkit *ProM*[2]. 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. Figs. 1-3 illustrate the effectiveness of our technique.

In the remainder of this paper, we first introduce preliminaries regarding Petri nets, partially ordered runs, and branching processes. Sect. 3 defines the operations for simplifying a mined process model illustrated by a running example. We report on experimental results in Sect. 4. Sect. 5 discusses related work and Sect. 6 concludes the paper.

2 Causal Behavior of Process Models w.r.t. a Log

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

2.1 Petri Nets and Logs

Initially, we assume some log L to be given, consisting of the cases $L = \{l_1, \ldots, l_n\}$ over the actions $\Sigma(L)$. Each case $l_i \in \Sigma(L)^*$ is a finite sequence of actions, $i = 1, \ldots, n$. In practice, a log may contain multiple identical cases; however we will not exploit this kind of information in the following. A mining algorithm $\mathcal{M}$ returns for a log L a Petri net $N = \mathcal{M}(L)$. In the following, we consider the case where (1) each action $t \in \Sigma(L)$ defines one transition t of N, and (2) the behavior of N *contains* all cases in L. That is, each $l \in L$ is an occurrence sequence of N.
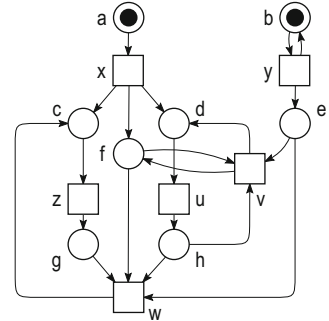

Fig. 5. A net system N

Definition 1 (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*. 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 mining 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

---

[2] ProM, including the new plug-in 'Uma', can be downloaded from www.processmining.org

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.

2.2   Partially Ordered Runs and Branching Processes

In the following, we study the behavior of N in terms of its *partially ordered* runs. We will use so-called *branching processes* 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:

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

- The places and transitions of π are *labeled* with the places and transitions of N, respectively.
- A place b of π with label p describes a *token* on p, the places of π with an empty pre-set describe the initial marking of N; b is called *condition*.
- A transition 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 $t^\bullet$; e is called *event*.
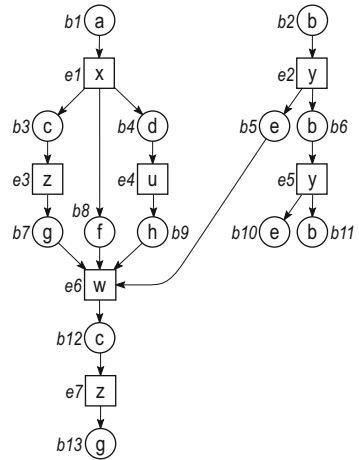
For example, event e_2 of π_1 in Fig. 6 describes an occurrence of y consuming token b_2 from place b and producing token b_5 on e and a *new* token b_6 on b. The events of π_1 are partially ordered: e_5 *depends on* e_2 whereas neither e_1 depends on e_2 nor e_2 on e_1. That is, e_1 and e_2 occur *concurrently*. The partially 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 distributed runs in a single structure; we will use it to calculate on 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 distributed run of N and we can read β as a special union of distributed 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 distributed 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*), λ assigns each *node* $x \in B \cup E$ a label.

Here, we give the constructive definition of the branching processes of N [12]. 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 $x_1 \parallel x_2$ 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 ($e_3 \parallel e_9$).

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

Base. Let $B_0 := \bigcup_{p \in P} \{b_p^1, \ldots, b_p^k \mid m_0(p) = k, \lambda(b_p^i) = p\}$ be a set of conditions *representing 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., $b_i \parallel b_j$, 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.

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

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 $\{b_1, \ldots, b_k\}$. 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 $\lambda'(c_i) = q_i$, $^\bullet c_i = \{e\}$ for $i = 1, \ldots, m$;
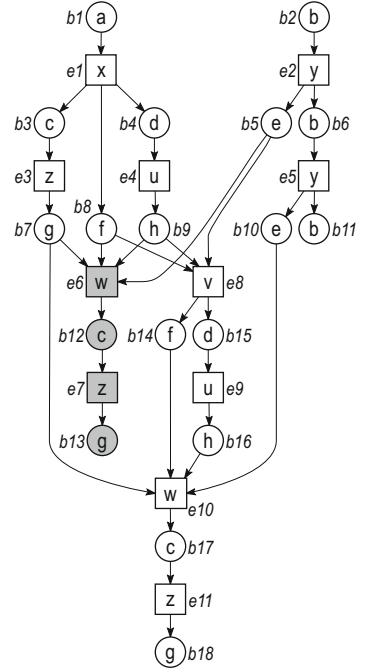
β' is a branching process of N. For example, assume the branching process β of Fig. 7 without $e_{10}, b_{17}, e_{11}, b_{18}$ to be given. The conditions $\{b_7, b_{14}, b_{16}, b_{10}\}$ are pair-wise concurrent and represent tokens in ${}^\bullet w$ of N of Fig. 5. Appending e_{10} (labeled w) and 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 [12]. 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 [13].

3   Operations to Simplify Process Models

Returning to our problem setting, we consider now the behavior of a Petri net $N = \mathcal{M}(L)$ that was discovered from a log L by a mining algorithm $\mathcal{M}$. Usually, each action in $\Sigma(L)$ becomes a transition of N. For the results of this paper, we assume that each case of L is an occurrence sequence of N i.e., each case of L can be replayed on N. For instance, the class of ILP-based mining algorithms returns nets of this kind [9].

As explained in Sect. 1, the net N can be structurally complex because of how $\mathcal{M}$ generalizes the behavioral information in L. In this section, we contribute a technique to balance the generalization done by $\mathcal{M}$ and simplify the structure of N.

The starting point to balance generalization is an *overfitting* net $N(L)$, which is derived from N and replays exactly each case of L (and all cases obtainable by reordering concurrent tasks of L.) We define several operations that *generalize* the behavior of $N(L)$ and *simplify its structure*. 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 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).

3.1   Starting Point: An Overfitting Model

The maximal branching process $\beta(N)$ of N introduced in Sect. 2.2 describes all behavior of N, not only the cases recorded in L. This additional behavior was introduced by the mining 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 this behavior.

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 $l = t_1 t_2 \ldots t_n \in L$. Initially, let $M := B_0$, $i := 1$.

1. Let $\{p_1, \ldots, p_k\} = {}^\bullet t_i$.
2. If there exists $\{b_1, \ldots, b_k\} \subseteq M$ with $\lambda(b_j) = p_j$, $j = 1, \ldots, k$ 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 $\{b_1, \ldots, b_k\}$; 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 $\{b_1, \ldots, b_k\} \subseteq M$ with $\lambda(b_j) = p_j$, $j = 1, \ldots, k$, and append a new event e, $\lambda(e) = t_i$ to all b_j, $j = 1, \ldots, k$, 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 $\{b_1, \ldots, b_k\}$ and compute the successor marking.]
4. $i := i + 1$, and return to step 1 if $i \leq n$.

This procedure is sound as N replays each $l \in L$. By construction, $\beta(L)$ is a smallest prefix of $\beta(N)$ that represents each $l \in L$. 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 $L = \{$xzuywz, xzuyvuywz, xzyuwz, xyzuvuywz, xuzywz, xuzyywz, yyxuvuzwz, ...$\}$.

$\beta(L)$ not only succinctly represents L, *but also all cases that differ from L by reordering concurrent actions*. The mining algorithm that returned N determines whether two actions are concurrent. Further, $\beta(L)$ already defines a Petri net that exhibits exactly 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 $\beta(L)$, i.e., $N(L)$ restricts the behavior of N to L.

3.2 Generalizing and Simplifying an Overfitting Model

The algorithm of the preceding section yields for a Petri net N discovered from a log L, an overfitting net $N(L)$ that exhibits exactly the branching process $\beta(L)$, i.e., the cases L. 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 executed by default 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.

(2) Then we structurally simplify the folded net by removing *implicit* places. An implicit place does not constrain whether a transition is enabled and hence can be removed [14]. 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.
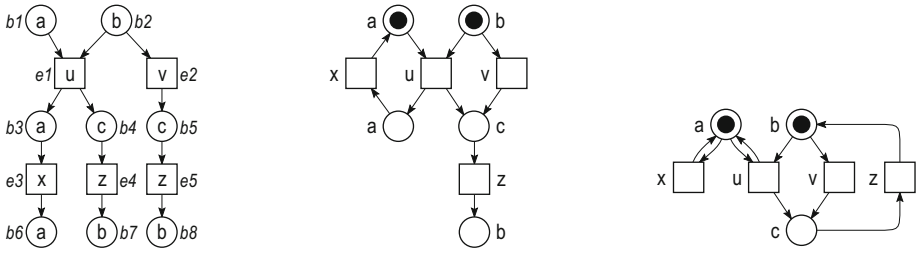
Fig. 8. The branching process β^2 (left) can be folded to different nets N_2 (middle) and N_2' (right) using different folding equivalences

3.3 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 2 (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. 3.4 will present a folding equivalence tailored towards process mining. Every folding equivalence of a branching process β induces a folded Petri net which is in principle the quotient of β under $\sim$.

Definition 3 (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* $b \in B_\beta$*.*

For example, on β^2 of Fig. 8 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 $\beta_\sim^2$ is N_2 of Fig. 8. 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. 8 (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.*
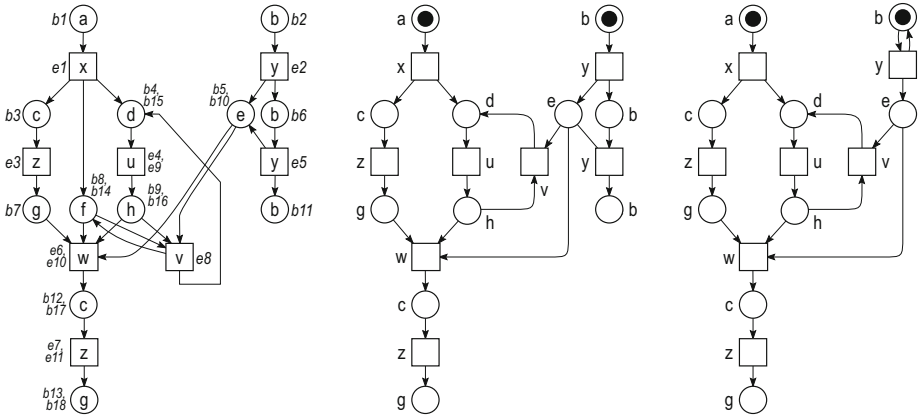
Fig. 9. Folding the branching process β of Fig. 7 by *future*(β) yields the Petri net $N_f(\beta)$ (left). Removing places of the implicit conditions b_8 and b_{14} yields the Petri net $N_i(\beta)$ (middle). Abstracting the chain of y transitions yields the net $N_c(\beta)$ (right).

Proof (Sketch). By Def. 2, 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 $\beta(N_2)$.

3.4   The Future Equivalence

The following procedure *future*(β) constructs a folding equivalence (Def. 2) 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. 3.1.

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 $b_1 \sim b_2$.
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. 9.
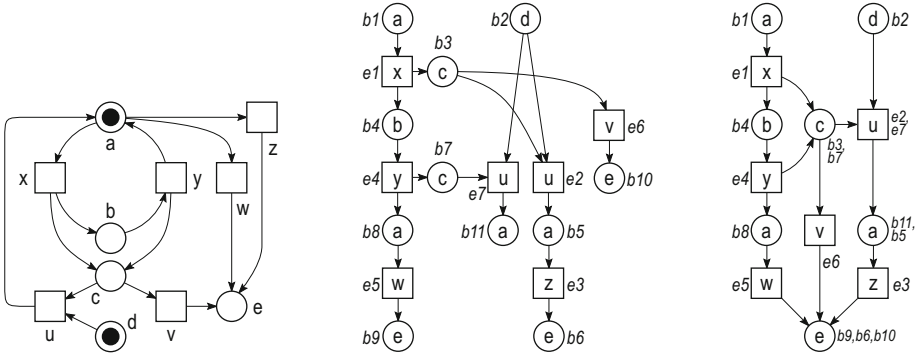
Fig. 10. The unsafe net N^3 (left) has among others the non-deterministic branching process β^3 (middle); a deterministic future equivalence merges transitions and results in a deterministic net $N_d(\beta^3) = \beta^3_{det(future(\beta^3))}$ (right)

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. 9. It differs from N of Fig. 5 primarily in representing action z twice in different contexts. This example illustrates the main effect of $future(\beta)$: to make process flow w.r.t. termination more explicit.

Complexitywise, $future(\beta)$ 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, $future(\beta)$ runs in $O(|E|^2 \cdot k)$ 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 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. 10 illustrates the situation.

The net N^3 of Fig. 10 and the log $L^3 = \{\text{xuzyw, xyvuw, xyuzw}\}$ yield the branching process $\beta^3 = \beta(N^3)$ shown in the middle. The future equivalence $future(\beta^3)$ 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 $det(\sim)$:
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 $det(future(\beta))$ 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 $future(\beta^3)$ of Fig. 10 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. 10 (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 inferred loops from β and merged nondeterministic transitions, which defers unobservable choices. For our running example (β in Fig. 7), $N_f(\beta)$ is already deterministic (cf. Fig. 9), 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 reduces the complexity of N by up to 30%.

3.5 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 4 (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 [14]. In the running example of Fig. 5, place f is implicit. This yields our second simplification operation: remove all implicit places from the folded net $N_d(\beta)$. Finding implicit places is a well-known problem and several techniques are applicable, e.g., [14].

Although being a standard technique, we learned from experiments that removing implicit places yields significant structural reduction on mined process models. In some cases, the structure simplified by up to 72%; up to 95% of the places were implicit.

3.6 Controlled Generalization of Process Models

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 mining algorithm $\mathcal{M}$ that returned N. Next, we present two operators to generalize N by *changing* N's structure.

Abstracting chains of unrestricted transitions. Mined Petri nets 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. 9.

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. 9 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 and complexity.

Splitting flower places. The last operation in this paper deals with a specific kind of places that are introduced by some mining algorithms and cannot be abstracted with the previous techniques.

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. 11 to the left: z may occur arbitrarily often before or after w, though only after x and before y occurred. While f in Fig. 11 certainly has an important function w.r.t. z, its effect on w is limited.

Fig. 11. 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.

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. 11 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 wrong. 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 mined process model. We observed structural simplification of up to 55% in experiments.
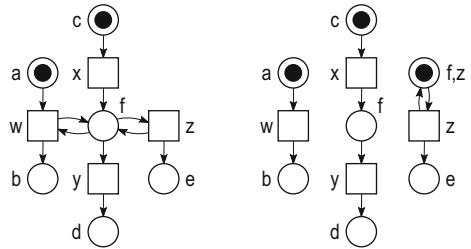
3.7   A Parameterized Process Model Simplification Algorithm

All operations presented in the preceding sections together yield the following algorithm for simplifying a mined Petri net $N = \mathcal{M}(L)$.

1. Construct the branching process $\beta(L)$ of N that represents all cases $l \in L$; construct the folding equivalence $\sim = det(future(\beta(L)))$; fold $N_d := \beta(L)_\sim$.

Table 1. Experimental results

	original	simplified	difference	runtime
	$|P|/|T|/|F|/c$	$|P|/|T|/|F|/c$	$|P|/|F|/c$	in sec
a22n0	21/ 22/ 61/ 1.41	21/ 22/ 55/ 1.27	0%/ -10%/ 1.11	1
a22n5	38/ 22/ 204/ 3.40	22/ 25/ 81/ 1.72	-42%/ -60%/ 1.98	4.1
a22n10	52/ 22/ 429/ 5.79	16/ 22/ 79/ 2.07	-69%/ -82%/ 2.80	89.2
a22n20	74/ 22/ 569/ 5.92	12/ 22/ 53/ 1.55	-84%/ -91%/ 3.82	389.5
a22n50	91/ 22/ 684/ 6.05	12/ 22/ 43/ 1.26	-87%/ -94%/ 4.80	639.7
a32n0	32/ 32/ 76/ 1.18	33/ 32/ 75/ 1.15	3%/ -1%/ 1.03	0.423
a32n5	44/ 32/ 228/ 3.00	33/ 32/ 96/ 1.47	-25%/ -58%/ 2.04	7.9
a32n10	68/ 32/ 543/ 5.43	20/ 32/ 89/ 1.71	-71%/ -84%/ 3.18	362.0
a32n20	90/ 32/ 613/ 5.02	25/ 32/ 87/ 1.52	-72%/ -86%/ 3.30	361.1
a32n50	110/ 32/ 868/ 6.11	20/ 32/ 84/ 1.61	-82%/ -90%/ 3.80	584.2
HC1	41/ 15/ 225/ 4.01	11/ 15/ 38/ 1.46	-73%/ -83%/ 2.75	0.119
HC2	20/ 14/ 140/ 4.11	12/ 16/ 39/ 1.39	-40%/ -72%/ 2.96	0.008
HC3	23/ 14/ 123/ 3.32	10/ 15/ 38/ 1.52	-57%/ -69%/ 2.18	0.004
HC4	43/ 17/ 224/ 3.73	11/ 17/ 45/ 1.60	-74%/ -80%/ 2.33	0.030
HC5	89/ 26/ 959/ 8.33	10/ 26/ 51/ 1.41	-89%/ -95%/ 5.91	0.298
M1	58/ 55/ 359/ 3.17	67/ 88/ 205/ 1.32	16%/ -43%/ 2.40	0.299
M2	31/ 23/ 256/ 4.74	16/ 23/ 55/ 1.41	-48%/ -79%/ 3.36	0.116

2. Remove implicit places from N_d.
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 [15,16]. Each of the four steps can be applied selectively. This way it is possible to balance between precision, generalization, and complexity reduction.

4   Experimental Results

We implemented our approach as a plugin for the process mining toolkit ProM. The user picks as input a log and a Petri net that was mined from this log, the plugin then shows a panel, where the user can configure the simplification of the net by disabling any of the steps as discussed in Sect. 3.7. By default, all steps are enabled and fully automatic requiring no Petri net knowledge from the user for model simplification.

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. 3.7 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 mined process model by removing up to 89% of the places, and up to 95% of the arcs.

Table 1 gives some more details. The logs named aXnY are benchmark logs having X different activities; the a$Xn0$ are logs of highly structured processes. Y is the percentage of random noise events introduced into the log a$Xn0$. 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 factor by which the graph complexity simplified, i.e., $c_{\text{difference}} = c_{\text{original}}/c_{\text{simplified}}$.

The numbers show that almost all models could be reduced significantly in terms of places and arcs (up to 87% and 94%). We observed that some models (a32n0, M1) grew slightly in size, i.e., more places and transitions were introduced. We found un-rolled loops of length greater than 2 that occur only once in the branching process to be responsible for the growth in size. Our algorithm cannot fold back these singular loops; though the algorithm could be extended to handle such patterns (see Sect. 3.6). Yet, even in case of larger nets, our technique reduced each Petri net's graph complexity c by a factor between 1.03 and 5.91 resulting in graph complexities of 1.15 to 2.07. 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.

We observed that splitting flower places is responsible for about 10% of the removed arcs in logs with a noise of 10% or more. Chain reduction removed between 12% and 51% of the transitions in logs with 5% noise and in M1. 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. Altogether, the case study processes demonstrate the feasibility of the technique in practice in terms of simplification and runtime.

5   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] and many algorithms have been proposed. Examples are the α algorithm [1], heuristic mining [3], genetic mining [5], fuzzy mining [4], etc. Of particular interest for this paper are the process discovery algorithms that guarantee a model with fitness 1 [8,7,6,9], e.g. several process mining techniques based on language-based regions have been proposed [8,9]. See [2] for a recent survey.

The approach of [10] 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 [10] as illustrated by Fig. 3.

Conformance checking techniques [16,15], like the post-processing approach presented in this paper, use a log and a model as input. In [15] the log is replayed on the Petri net to see where the model and reality diverge. In [16] 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.

The goal of this paper is to simplify and structure discovered process models. This is related to techniques transforming unstructured processes models in structured models [17,18]. However, these techniques do not consider the real observed behavior.

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., [19]. However, these approaches require N to have a finite state space, which usually does not hold for mined process models. Additionally, our aim is also to structurally simplify N, not only to restrict it to L.

6   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.

References

1. van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer, Heidelberg (2011)
2. van Dongen, B., de Medeiros, A.A., Wen, L.: Process Mining: Overview and Outlook of Petri Net Discovery Algorithms. ToPNOC 2, 225–242 (2009)
3. Weijters, A., van der Aalst, W.: Rediscovering Workflow Models from Event-Based Data using Little Thumb. Integrated Computer-Aided Engineering 10(2), 151–162 (2003)
4. Günther, C., van der Aalst, W.: Fuzzy Mining: Adaptive Process Simplification Based on Multi-perspective Metrics. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 328–343. Springer, Heidelberg (2007)
5. Medeiros, A., Weijters, A., van der Aalst, W.: Genetic Process Mining: An Experimental Evaluation. Data Mining and Knowledge Discovery 14(2), 245–304 (2007)
6. van Dongen, B., van der Aalst, W.: Multi-Phase Process Mining: Building Instance Graphs. In: Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T.-W. (eds.) ER 2004. LNCS, vol. 3288, pp. 362–376. Springer, Heidelberg (2004)
7. Carmona, J., Cortadella, J.: Process Mining Meets Abstract Interpretation. In: Balcázar, J.L., Bonchi, F., Gionis, A., Sebag, M. (eds.) ECML PKDD 2010. LNCS, vol. 6321, pp. 184–199. Springer, Heidelberg (2010)
8. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process Mining Based on Regions of Languages. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 375–383. Springer, Heidelberg (2007)

9. van der Werf, J., van Dongen, B., Hurkens, C., Serebrenik, A.: Process Discovery using Integer Linear Programming. Fundamenta Informaticae 94, 387–412 (2010)
10. van der Aalst, W., Rubin, V., Verbeek, H., van Dongen, B., Kindler, E., Günther, C.W.: Process mining: a two-step approach to balance between underfitting and overfitting. Software and System Modeling 9(1), 87–111 (2010)
11. Adriansyah, A., van Dongen, B., van der Aalst, W.: Towards Robust Conformance Checking. In: Muehlen, M., Su, J. (eds.) TEMPO. LNBIP, vol. 66, pp. 122–133. Springer, Heidelberg (2011)
12. Esparza, J., Römer, S., Vogler, W.: An Improvement of McMillan's Unfolding Algorithm. Formal Methods in System Design 20(3), 285–310 (2002)
13. Engelfriet, J.: Branching Processes of Petri Nets. Acta Informatica 28(6), 575–591 (1991)
14. Colom, J., Silva, M.: Improving the Linearly Based Characterization of P/T Nets. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 113–145. Springer, Heidelberg (1991)
15. Rozinat, A., van der Aalst, W.: Conformance Checking of Processes Based on Monitoring Real Behavior. Information Systems 33(1), 64–95 (2008)
16. Muñoz-Gama, J., Carmona, J.: A Fresh Look at Precision in Process Conformance. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 211–226. Springer, Heidelberg (2010)
17. Polyvyanyy, A., García-Bañuelos, L., Dumas, M.: Structuring Acyclic Process Models. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 276–293. Springer, Heidelberg (2010)
18. Vanhatalo, J., Völzer, H., Koehler, J.: The Refined Process Structure Tree. Data Knowledge Engineering 68(9), 793–818 (2009)
19. Lüder, A., Hanisch, H.: Synthesis of Admissible Behavior of Petri Nets for Partial Order Specifications. In: WODES 2000, pp. 409–431. Kluwer, Dordrecht (2000)