Discovering Directly-Follows Complete Petri Nets From Event Data

Wil M.P. van der Aalst

Process and Data Science (PADS), RWTH Aachen University, Germany
wvdaalst@pads.rwth-aachen.de www.vdaalst.com

Abstract. Process mining relies on the ability to discover high-quality process models from event data describing only example behavior. Process discovery is challenging because event data only provide positive examples and process models may serve different purposes (performance analysis, compliance checking, predictive analytics, etc.). This paper focuses on the *discovery* of *accepting Petri nets* under the assumption that both the event log and process model are *directly-follows complete*. Based on novel insights, two new variants ($\alpha^{1.1}$ and $\alpha^{2.0}$) of the well-known *Alpha* algorithm ($\alpha^{1.0}$) are proposed. These variants overcome some of the limitations of the classical algorithm (e.g., dealing with short-loops and non-unique start and ending activities) and shed light on the boundaries of the "directly-follows completeness" assumption. These insights can be leveraged to create new process discovery algorithms or improve existing ones.

Keywords: Process Discovery · Process Models · Petri Nets

1 Introduction

Process mining is increasingly adopted by larger organizations to find and remove inefficiencies, bottlenecks, and compliance issues [1]. There are over 40 process mining vendors (cf. www.processmining.org) and more than half of the Fortune-500 corporations are already using process mining [16]. Thousands of organizations are extracting event data from systems such SAP, Salesforce, Oracle, ServiceNow, and Workday to apply process mining. Despite the widespread use of process mining, many challenges remain, ranging from data extraction and scalability to discovering better process models and providing better diagnostics.

Although most process mining tools support the discovery of higher-level models visualized in terms of BPMN (Business Process Model and Notation) (next to conformance checking and predictive analytics), in practice process analysts and managers mostly use the so-called *Directly-Follows Graphs* (DFGs) to get initial insights. In a DFG all activities and their frequencies are shown. The activities are connected through directed edges that show how often one activity is followed by another activity within the same case (i.e., process instance). These edges can also be annotated with durations (minimum, maximum, mean, etc.), because events have timestamps. The creation of DFGs is simple and highly scalable. However, there are also many limitations (e.g., producing complex underfitting process models), as discussed in [2].

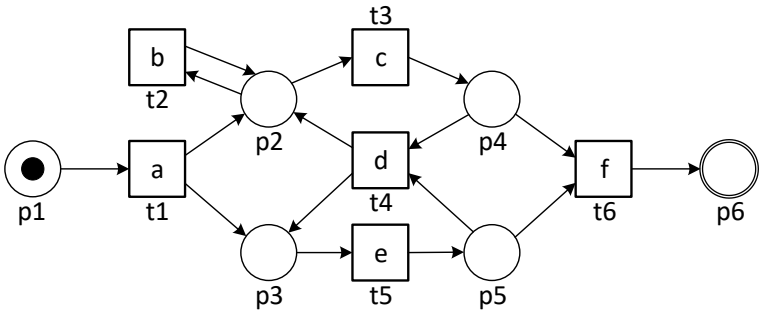


Fig. 1. Accepting Petri net AN_1 discovered for $L_1 = [\langle a,c,e,f\rangle^5, \langle a,e,c,f\rangle^4, \langle a,b,c,e,f\rangle^4, \langle a,b,e,c,f\rangle^3, \langle a,e,b,c,f\rangle^3, \langle a,b,b,c,e,f\rangle^3, \langle a,b,b,e,c,f\rangle^2, \langle a,e,b,b,c,f\rangle^2, \langle a,c,e,d,c,e,f\rangle^2, \langle a,e,c,d,c,e,f\rangle^2, \langle a,c,e,d,b,c,e,f\rangle^2, \langle a,e,c,d,b,b,c,e,f\rangle^2, \langle a,b,c,e,d,c,e,f\rangle, \langle a,b,e,c,d,e,c,f\rangle, \langle a,e,b,c,d,c,e,f\rangle, \langle a,b,c,e,d,e,c,f\rangle, \langle a,b,c,e,d,c,e,d,c,e,d,b,e,b,c,f\rangle, \langle a,c,e,d,c,e,d,e,c,f\rangle]$ using the new Alpha algorithm.

To overcome the limitations of DFGs, dozens (if not hundreds) of process discovery techniques have been developed. The core idea is very simple. The input (i.e., an event log) can be viewed as a *multiset of traces*. Each trace corresponds to a case, i.e., one execution of the process for a patient, order, student, package, etc. A trace is represented as a sequence of activities. Since multiple cases may exhibit the same sequence, we need to consider multisets. $L_1 = [\langle a,c,e,f\rangle^5, \langle a,e,c,f\rangle^4, \ldots]$ in the caption of Figure 1 describes such an event log, e.g., there are five cases following the sequence $\langle a,c,e,f\rangle$. Note that in this compact representation, we abstract from timestamps, resources, costs, etc. Based on such input, we would like to produce models such as the accepting Petri net AN_1 shown in Figure 1. AN_1 is able to produce all the traces in L_1. Note that activity e is concurrent to b and c such that c and e occur the same number of times (at least once) and b can occur any number of times.

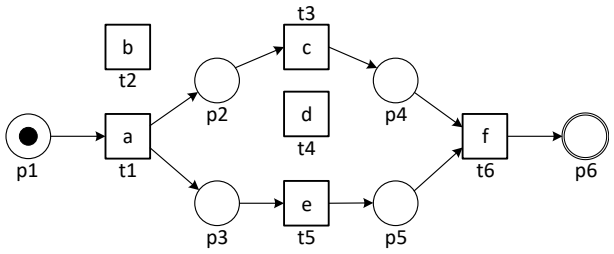


Fig. 2. Accepting Petri net AN_1' discovered for L_1 using the classical Alpha algorithm.

The original *Alpha* algorithm [1, 5] was the first algorithm able to discover concurrent process models from event logs. The algorithm assumes that the underlying process can be represented as a free-choice structured workflow net without short loops.

For this class of process models, the algorithm guarantees to return the correct process model assuming a directly-follows complete event log [5]. For process models outside this class, the results are unpredictable. The discovered model may be underfitting or not sound. The limitations were already investigated in the original paper. One of the problems is the inability to discover short loops. This is illustrated in Figure 2, which shows the process model when applying the original, over twenty year old, Alpha algorithm. The self-loop involving b and the length-two loops involving d cause problems. Activities b and d are disconnected and it is impossible to replay parts of the event log L_1. The Alpha algorithm has more problems, some of which can be resolved by using more information [24, 25]. However, these extensions are complex and impose more assumptions.

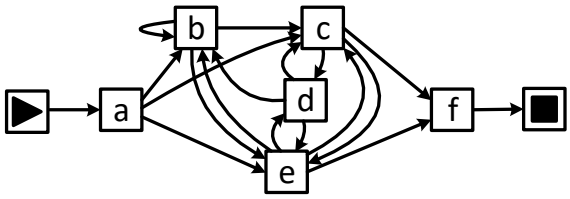


Fig. 3. Directly-Follows Graph (DFG) based on L_1.

In this paper, we take a different route. We simply assume that the Directly-Follows Graph (DFG) is all the information we have. For event log L_1, we assume that the DFG in Figure 3 is all we have. Moreover, we look beyond the traditional subclasses of Petri nets. We do not assume workflow nets or the free-choice property. Instead, we focus on *structural directly-follows complete* accepting Petri nets. The accepting Petri net AN_1 shown in Figure 1 is an example of this class of models. Therefore, we consider both directly-follows complete event logs and structural directly-follows complete process models. This provides novel insights and also leads to *two new versions of the Alpha algorithm* that are as compact and simple as the original algorithm, but more powerful.

The remainder of this paper is organized as follows. Section 2 introduces event logs and accepting Petri nets. Section 3 discusses different notions of directly-follows completeness and Section 4 lists the typical subclasses of Petri nets considered. Section 5 presents an improved version of the original algorithm that drops the workflow net assumption. Section 6 changes the core part of the algorithm allowing for the discovery of short loops. The paper ends with a discussion (Section 7) and conclusion (Section 8).

2 Preliminaries

In this section, we introduce basic concepts, but assume that the reader is (somewhat) familiar with the basics of Petri nets and process mining. For completeness, we refer to [1] for process mining and [13, 14] for an extensive introduction to Petri nets.

In process mining, multisets and sequences play an important role. $\mathcal{B}(X) = X \to \mathbb{N}$ is a multiset over X. For example, if $X = \{a, b, c\}$, then $[a^3, b^2, c] \in \mathcal{B}(X)$, $[a^4, b^2] \in$

$\mathcal{B}(X)$, $[a^6] \in \mathcal{B}(X)$ are three multisets each consisting of six elements. X^* is the set of sequences over X, e.g., $\sigma = \langle a, b, a, b, c \rangle \in X^*$. We use the standard operations on multisets and sequences, e.g., $\{x \in \sigma\} = \{a, b, c\}$ and $[x \in \sigma] = [a^2, b^2, c]$.

We start by introducing *event logs*. An event log is a collection of events typically grouped in *cases*, ordered by *time*, and labeled by *activity names*. In its simplest form each event has a case identifier, an activity name, and a timestamp. Events can have many more attributes, e.g., costs, resource, location, etc. There are also more sophisticated event log notions, e.g., partially ordered events, events with explicit uncertainty, and events that may refer to any number of objects of different types, see, for example, the eXtensible Event Stream (XES, `www.xes-standard.org`) standard and Object Centric Event Log (OCEL, `www.ocel-standard.org`) standard. Here, we consider the most basic setting and only consider the ordering of activities within cases. This implies that an event log can be described as a *multiset of traces*, where each trace is a *sequence of activities* (also called a *process variant*). An example trace is $\sigma = \langle a, b, a \rangle$. Many cases can have the same trace. Therefore, an event log is a multiset of traces. E.g. $L = [\langle a, b, a \rangle^{12}, \langle b, a, a \rangle^8]$ is an event log with 60 events describing the traces of 20 cases distributed over two process variants.

Definition 1 (Event Log). *$\mathcal{U}_{act}$ is the universe of activity names. A trace $\sigma = \langle a_1, a_2, \ldots, a_n \rangle \in {\mathcal{U}_{act}}^*$ is a sequence of activities. An event log $L \in \mathcal{B}({\mathcal{U}_{act}}^*)$ is a multiset of traces.*

In the caption of Figure 1, there is another (larger) example of an event log $L_1 = [\langle a, c, e, f \rangle^5, \langle a, e, c, f \rangle^4, \langle a, b, c, e, f \rangle^4, \langle a, b, e, c, f \rangle^3, \ldots]$, e.g., there are five cases corresponding to trace $\langle a, c, e, f \rangle$, four cases corresponding to $\langle a, e, c, f \rangle$, etc.

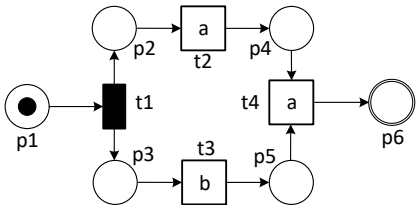


Fig. 4. An accepting Petri net $AN_2 = (N, M_{init}, M_{final})$ with $M_{init} = [p1]$, $M_{final} = [p6]$, and $lang(AN_2) = \{\langle a, b, a \rangle, \langle b, a, a \rangle\}$.

We use *accepting Petri nets* to model processes. Figure 4 shows an example accepting Petri net AN_2 allowing for two traces: $\langle a, b, a \rangle$ and $\langle b, a, a \rangle$. The Petri net has six places $P = \{p1, p2, p3, p4, p5, p6\}$ (represented by circles), four transitions $T = \{t1, t2, t3, t4\}$ (represented by squares), and 10 arcs $F = \{(p1, t1), (t1, p2), \ldots, (t4, p6)\}$. Transitions can be labeled, e.g., $l(t2) = a$ and $l(t3) = b$.

Definition 2 (Labeled Petri Net). *A labeled Petri net is a tuple $N = (P, T, F, l)$ with P the set of places, T the set of transitions, $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ the flow relation, and $l \in T \not\to \mathcal{U}_{act}$ a labeling function. $\bullet t = \{p \in P \mid (p, t) \in F\}$ is*

the set of input places and $t\bullet = \{p \in P \mid (t,p) \in F\}$ *is the set of output places of a transition* $t \in T$*. The same notation can be used for input and output transitions of a place* $p \in P$*:* $\bullet p = \{t \in T \mid (t,p) \in F\}$ *and* $p\bullet = \{t \in T \mid (p,t) \in F\}$*.*

Note that the labeling function may be partial and that multiple transitions may have the same label. For AN_2 in Figure 4, $t1 \notin dom(l)$ and $l(t2) = l(t4)$. It a transition t has no label, i.e., $t \notin dom(l)$, we also write $l(t) = \tau$ and say the transition is *silent* or *invisible*. In Figure 4, $\bullet p1 = \emptyset$, $p1\bullet = \{t1\}$, $\bullet t1 = \{p1\}$, $t1\bullet = \{p2, p3\}$, etc. States in Petri nets are called *markings* that mark certain places with *tokens* (represented by black dots). Technically, a marking is a multiset of places $M \in \mathcal{B}(P)$. An *accepting* Petri net has an *initial marking* M_{init} and a *final marking* M_{final}.

Definition 3 (Accepting Petri Net). *An accepting Petri net is a triplet* $AN = (N, M_{init}, M_{final})$ *where* $N = (P, T, F, l)$ *is a labeled Petri net,* $M_{init} \in \mathcal{B}(P)$ *is the initial marking, and* $M_{final} \in \mathcal{B}(P)$ *is the final marking.* $\mathcal{U}_{AN}$ *is the set of accepting Petri nets.*

In AN_2 depicted in Figure 4, $M_{init} = [p1]$ is the initial marking, and $M_{final} = [p6]$. A transition is called *enabled* if each of the input places has a token. An enabled transition may *fire* (i.e., occur), thereby consuming a token from each input place and producing a token for each output place. For example, firing $t1$ in the initial making leads to marking $[p2, p3]$. There are 6 reachable markings starting from $M_{init} = [p1]$: $[p1]$, $[p2, p3]$, $[p2, p5]$, $[p3, p4]$, $[p4, p5]$, and $[p6]$.

Definition 4 (Reachable Markings and Enabled Firing Sequences). *Let* $AN = (N, M_{init}, M_{final}) \in \mathcal{U}_{AN}$ *be an accepting Petri net with* $N = (P, T, F, l)$*.* $M_1 \xrightarrow{t} M_2$ *denotes that in* $M_1 \in \mathcal{B}(P)$ *transition* $t \in T$ *is enabled and firing* t *results in marking* $M_2 \in \mathcal{B}(P)$*.* $M_1 \xrightarrow{\sigma} M_n$ *with* $\sigma = \langle t_1, t_2, \ldots t_{n-1} \rangle \in T^*$ *denotes that there are markings* $M_2, \ldots M_{n-1} \in \mathcal{B}(P)$ *such that* $M_i \xrightarrow{t_i} M_{i+1}$ *for* $1 \leq i < n$*, i.e., there is an enabled firing sequence* σ *leading from* M_1 *to* M_n*.* $efs(AN) = \{\sigma \in T^* \mid \exists_{M \in \mathcal{B}(P)}\ M_{init} \xrightarrow{\sigma} M\}$ *is the set of enabled firing sequences.* $rmk(AN) = \{M \in \mathcal{B}(P) \mid \exists_{\sigma \in T^*}\ M_{init} \xrightarrow{\sigma} M\}$ *is the set of reachable markings.*

Definition 5 (Liveness and Boundedness). *Let* $AN = (N, M_{init}, M_{final}) \in \mathcal{U}_{AN}$ *be an accepting Petri net with* $N = (P, T, F, l)$*.* AN *is live if for any reachable marking* $M \in rmk(AN)$ *and every transition* $t \in T$*, there exists a marking reachable from* M *enabling* t*.* AN *is bounded if* $rmk(AN)$ *is finite (i.e., there is a* k *such that no place can have more than* k *tokens).* AN *is safe if for any* $M \in rmk(AN)$ *and* $p \in P$*:* $M(p) \leq 1$ *(i.e., each place holds at most 1 token in any reachable marking).*

Accepting Petri net AN_1 in Figure 1 has six reachable markings and is safe, but not live. AN_2 depicted in Figure 4 also has six reachable markings and is safe, but not live. Adding a short-circuiting transition t_{sc} connecting $p6$ to $p1$ (i.e., $\bullet t_{sc} = \{p6\}$ and $t_{sc}\bullet = \{p1\}$) in figures 1 and 4 makes both nets live.

Definition 6 (Complete Firing Sequences). *Let* $AN = (N, M_{init}, M_{final}) \in \mathcal{U}_{AN}$ *be an accepting Petri net with* $N = (P, T, F, l)$*.* $cfs(AN) = \{\sigma \in T^* \mid M_{init} \xrightarrow{\sigma} M_{final}\}$ *is the set of complete firing sequences of* AN*, i.e., all firing sequences starting in the initial marking* M_{init} *and ending in the final marking* M_{final}*.*

For the accepting Petri net AN_2 in Figure 4, $cfs(AN_2) = \{\langle t1, t2, t3, t4\rangle, \langle t1, t3, t2, t4\rangle\}$. The accepting Petri net AN_1 in Figure 1, allows for arbitrary long complete firing sequences. Hence, $cfs(AN_1) = \{\langle t1, t3, t5, t6\rangle, \langle t1, t2, t3, t5, t6\rangle, \langle t1, t2, t5, t2, t2, t3, t6\rangle, \ldots\}$ has infinitely many elements.

Firing a transition t corresponds to executing activity $l(t)$ if $t \in dom(l)$. To map complete firing sequences to traces, we apply labeling function l such that visible transitions are mapped onto activities and visible transitions are skipped. For AN_2, i.e., the accepting Petri net in Figure 4, and the two corresponding complete firing sequences $\sigma_1 = \langle t1, t2, t3, t4\rangle$ and $\sigma_2 = \langle t1, t3, t2, t4\rangle$: $l(\sigma_1) = \langle a, b, a\rangle$ and $l(\sigma_2) = \langle b, a, a\rangle$. Note that firing $t1$ cannot be observed and $t2$ and $t4$ are mapped onto the same activity a.

Definition 7 (Traces of an Accepting Petri Net). *Let $AN = (N, M_{init}, M_{final}) \in \mathcal{U}_{AN}$ be an accepting Petri net. $lang(AN) = \{l(\sigma) \mid \sigma \in cfs(AN)\}$ are the traces possible according to AN.*

For the accepting Petri net AN_2 in Figure 4, $lang(AN_2) = \{\langle a, b, a\rangle, \langle b, a, a\rangle\}$. The accepting Petri net AN_1 in Figure 1, allows for infinitely many traces: $lang(AN_1) = \{\langle a, c, e, f\rangle, \langle a, b, c, e, f\rangle, \langle a, b, e, b, b, c, f\rangle, \ldots\}$.

An accepting Petri net is *sound*, if there are no dead transitions and, from any reachable state, it is possible to reach the final state.

Definition 8 (Soundness). *Let $AN = (N, M_{init}, M_{final}) \in \mathcal{U}_{AN}$ be an accepting Petri net with $N = (P, T, F, l)$. AN is sound if and only if (1) for any $t \in T$ there exists a $\sigma \in efs(AN)$ such that $t \in \sigma$ (i.e., t is not dead), and (2) for any $\sigma_1 \in efs(AN)$ there exists a $\sigma_2 \in T^*$ such that $\sigma_1 \cdot \sigma_2 \in cfs(AN)$ (i.e., from any reachable marking, it is possible to reach the final state).*

Discovering a process model from a collection of example traces is one of the main process mining tasks. Ideally, the discovered process model is sound.

Definition 9 (Process Discovery). *A discovery algorithm $disc \in \mathcal{B}(\mathcal{U}_{act}{}^*) \rightarrow \mathcal{U}_{AN}$ produces an accepting Petri net for each event log.*

Many algorithms described in literature implement a discovery function $disc \in \mathcal{B}(\mathcal{U}_{act}{}^*) \rightarrow \mathcal{U}_{AN}$, e.g., [4, 5, 9–11, 20, 17–19, 21, 24–27]. Not all are explicitly discovering accepting Petri nets. However, also process trees can be converted into accepting Petri nets and if an explicit final marking is missing it can often be added.

Definition 10 (Conformance Checking). *Let $\mathcal{U}_{diag}$ be the universe of conformance diagnostics. A conformance checking algorithm $conf \in \mathcal{B}(\mathcal{U}_{act}{}^*) \times \mathcal{U}_{AN} \rightarrow \mathcal{U}_{diag}$ produces conformance diagnostics given an event log and accepting Petri net as input.*

Conformance checking is a topic in itself [1, 3, 12, 22]. Therefore, we do not detail the type of diagnostics $\mathcal{U}_{diag}$. Most conformance measures are normalized to $[0, 1]$ where values close to 0 are bad and values close to 1 are good.

Conformance diagnostics may focus on (1) *recall*, also called (replay) fitness, which aims to quantify the fraction of observed behavior that is allowed by the model, (2)

precision, which aims to quantify the fraction of behavior allowed by the model that was actually observed (i.e., avoids "underfitting" the event data), (3) *generalization*, which aims to quantify the probability that new unseen cases will fit the model (i.e., avoids "overfitting" the event data), and (4) *simplicity*, which refers to Occam's Razor and can be made operational by quantifying the complexity of the model (number of nodes, number of arcs, understandability, etc.).

Let's try to operationalize recall and precision. Recall is concerned with traces in the event log not possible in the model, i.e., $L_{nofit} = [\sigma \in L \mid \sigma \notin lang(AN)]$. Precision is concerned with traces possible in the model, but not appearing in the event log, i.e., $L_{miss} = \{\sigma \in lang(AN) \mid \sigma \notin L\}$. However, this is not so simple. The event log contains only example behavior (a sample) and any model with loops has infinitely many traces. In such cases L_{miss} has infinitely many elements by definition. If the model aims to describe the mainstream behavior, then L_{nofit} may contain exceptional behavior that was left out deliberately. Moreover, traces may be partly fitting and one often wants to strike a balance between precision (avoiding "underfitting" the sample event data) and generalization (avoiding "overfitting" the sample event data).

These considerations make *process mining different from many other model-learning techniques* such as synthesis, system identification, grammatical inference, regular inference, automata learning [6–8, 15, 23]. The field of model learning can be structured using three dimensions: (a) only positive examples versus positive and negative examples, (b) input data is complete (in some form) or not, (c) passive learning (just observations) versus active learning (interactions). Process mining focuses on passive learning using only positive examples with only weak completeness guarantees. This makes it very difficult. For example, it is impossible to actively test hypotheses.

A detailed discussion of process discovery and conformance checking techniques (including possible quality criteria) is outside the scope of this paper (see [1, 3, 12, 22] for details).

3 Directly-Follows Completeness

Most process discovery algorithms heavily rely on the *directly-follows relation*, i.e., activities following each other directly, either in the event log or process model. The goal is to find models that have the same directly-follows relation as seen in the event log. The motivation to do this is simple. *One cannot expect to see all possible traces in an event log.* The event log only contains a sample. However, it is reasonable to assume that one can witness the complete directly-follows relation in the event log, i.e., if a can be followed by b we should see it at least once in the input data.

In the section, we define different *directly-follows completeness* notions, also considering the structure of the accepting Petri net.

Consider event log $L_1 = [\langle a, c, e, f\rangle^5, \langle a, e, c, f\rangle^4, \langle a, b, c, e, f\rangle^4, \langle a, b, e, c, f\rangle^3, \langle a, e, b, c, f\rangle^3, \langle a, b, b, c, e, f\rangle^3, \langle a, b, b, e, c, f\rangle^2, \langle a, e, b, b, c, f\rangle^2, \langle a, c, e, d, c, e, f\rangle^2, \langle a, e, c, d, c, e, f\rangle^2, \langle a, c, e, d, b, c, e, f\rangle^2, \langle a, e, c, d, b, b, c, e, f\rangle^2, \langle a, b, c, e, d, c, e, f\rangle, \langle a, b, e, c, d, e, c, f\rangle, \langle a, e, b, c, d, c, e, f\rangle, \langle a, b, c, e, d, e, c, f\rangle, \langle a, b, c, e, d, c, e, d, c, e, d, b, e, b, c, f\rangle, \langle a, c, e, d, c, e, d, e, c, f\rangle]$ and accepting Petri net AN_1 shown in Figure 1. Assume that L_1 was obtained by repeatedly *simulating* the accepting Petri net AN_1. In this case,

we have a *known ground truth*, because we want the discovery algorithm $disc$ to discover AN_1 based on L_1, i.e., $disc(L_1) = AN_1$. However, due to the two loops and concurrency, there are many possible traces. The *self-loop* involving b and the *length-two loops* involving d allow for an arbitrary number of b's, c's, d's, and e's in a single trace. Hence, one *cannot* expect to observe all possible traces. In fact, this is impossible. One can try to increase the sample (i.e., the number traces in the event log), but the foundational problem remains: we only have example traces. However, event log L_1 is *directly-follows complete* with respect to model AN_1. This is reflected by the *Directly-Follows Graph* (DFG) in Figure 3. To explain directly-follows completeness, we first define some core concepts.

Definition 11 (Adding Artificial Start and End Activities). *$\blacktriangleright \notin \mathcal{U}_{act}$ is an artificial start activity, $\blacksquare \notin \mathcal{U}_{act}$ is an artificial end activity, $\hat{\mathcal{U}_{act}} = \mathcal{U}_{act} \cup \{\blacktriangleright, \blacksquare\}$. For any $\sigma \in {\mathcal{U}_{act}}^*$, $\hat{\sigma} = \langle\blacktriangleright\rangle \cdot \sigma \cdot \langle\blacksquare\rangle$. For any $L \in \mathcal{B}({\mathcal{U}_{act}}^*)$, $\hat{L} = [\hat{\sigma} \mid \sigma \in L]$. For any $S \subseteq {\mathcal{U}_{act}}^*$, $\hat{S} = \{\hat{\sigma} \mid \sigma \in S\}$.*

The "hat notation" adds artificial starts and ends to traces, languages, and event logs. For $L_2 = [\langle a, b, a\rangle^5, \langle b, a, a\rangle^4]$: $\hat{L_2} = [\langle\blacktriangleright, a, b, a, \blacksquare\rangle^5, \langle\blacktriangleright, b, a, a, \blacksquare\rangle^4]$.

Definition 12 (Log-Based Directly-Follows Relation). *Let $L \in \mathcal{B}({\mathcal{U}_{act}}^*)$ be an event log.*

- *$act(L) = [\sigma(i) \mid \sigma \in L \wedge 1 \leq i \leq |\sigma|]$ is the multiset of activities in the event log (note that $\sigma(i)$ is the i-th element in the sequence σ).*
- *$df(L) = [(\sigma(i), \sigma(i+1)) \mid \sigma \in \hat{L} \wedge 1 \leq i < |\sigma|]$ is the multiset of directly-follows relations in the event log (note that the artificial start activity $\blacktriangleright$ and end activity $\blacksquare$ have been added to the traces in $\hat{L}$).*
- *$a_1 \Rightarrow_L a_2$ if and only if $(a_1, a_2) \in df(L)$, $a_1 \not\Rightarrow_L a_2$ if and only if $(a_1, a_2) \notin df(L)$, and $a_1 \Leftrightarrow_L a_2$ if and only if $a_1 \Rightarrow_L a_2$ and $a_2 \Rightarrow_L a_1$.*

Take again $L_2 = [\langle a, b, a\rangle^5, \langle b, a, a\rangle^4]$: $act(L_2) = [a^{18}, b^9]$, $df(L_2) = [(\blacktriangleright, a)^5, (\blacktriangleright, b)^4, (a, a)^4, (a, b)^5, (b, a)^9, (a, \blacksquare)^9]$. Hence, we can write $\blacktriangleright \Rightarrow_{L_2} a$, $b \Rightarrow_{L_2} a$, $b \not\Rightarrow_{L_2} b$, $a \Rightarrow_{L_2} \blacksquare$, etc. Similar relations can be obtained for process models as is defined next.

Definition 13 (Behavioral Model-Based Directly-Follows Relation). *Let $AN = (N, M_{init}, M_{final}) \in \mathcal{U}_{AN}$ be an accepting Petri net with $S = lang(AN)$ as possible traces.*

- *$act^b(AN) = \{\sigma(i) \mid \sigma \in S \wedge 1 \leq i \leq |\sigma|\}$ is the set of activities possible according to the model's behavior.*
- *$df^b(AN) = \{(\sigma(i), \sigma(i+1)) \mid \sigma \in \hat{S} \wedge 1 \leq i < |\sigma|\}$ is the set of directly-follows relations possible according to the model's behavior.*
- *$a_1 \Rightarrow^b_{AN} a_2$ if and only if $(a_1, a_2) \in df^b(AN)$, $a_1 \not\Rightarrow^b_{AN} a_2$ if and only if $(a_1, a_2) \notin df^b(AN)$, and $a_1 \Leftrightarrow^b_{AN} a_2$ if and only if $a_1 \Rightarrow^b_{AN} a_2$ and $a_2 \Rightarrow^b_{AN} a_1$.*

For the accepting Petri net AN_2 in Figure 4, we find $act^b(AN_2) = \{a, b\}$, $df^b(AN_2) = \{(\blacktriangleright, a), (\blacktriangleright, b), (a, a), (a, b), (b, a), (a, \blacksquare)\}$. Hence, we can write $\blacktriangleright \Rightarrow^b_{AN_2} a$, $b \Rightarrow^b_{AN_2} a$, $b \not\Rightarrow^b_{AN_2} b$, $a \Rightarrow^b_{AN_2} \blacksquare$, etc.

Next, we consider a novel concept that takes the *structure* of the accepting Petri net into account. For this, we only consider sound models where all transitions have a label (no silent transitions). Each place has a set of input and output transitions. These are in a *structural* directly-follows relation. If $t_1 \in \bullet p$ and $t_2 \in p\bullet$, then we expect – based on the structure – that activity $l(t_1)$ can be directly followed by activity $l(t_2)$. If multiple places in the Petri net can be enabled concurrently, the same is expected to hold for the input transitions of one place and the output transitions of another concurrently marked place.

Definition 14 (Structural Model-Based Directly-Follows Relation). *Let $AN = (N, M_{init}, M_{final}) \in \mathcal{U}_{AN}$ be a sound accepting Petri net with $N = (P, T, F, l)$ and $dom(l) = T$.*

- *$act^s(AN) = \{l(t) \mid t \in T\}$ is the set of activities in the model (consider only structure and not behavior).*
- *$df^s(AN) = \{(l(t_1), l(t_2)) \mid \exists_{M \in rmk(AN)} \exists_{p_1, p_2 \in M}\ t_1 \in \bullet p_1 \ \wedge\ t_2 \in p_2\bullet\} \cup \{(\blacktriangleright, l(t)) \mid \exists_{p \in M_{init}}\ t \in p\bullet\} \cup \{(l(t), \blacksquare) \mid \exists_{p \in M_{final}}\ t \in \bullet p\}$ is the set of directly-follows relations possible according to the model's structure.*
- *$a_1 \Rightarrow^s_{AN} a_2$ if and only if $(a_1, a_2) \in df^s(AN)$, $a_1 \not\Rightarrow^s_{AN} a_2$ if and only if $(a_1, a_2) \notin df^s(AN)$, and $a_1 \Leftrightarrow^s_{AN} a_2$ if and only if $a_1 \Rightarrow^s_{AN} a_2$ and $a_2 \Rightarrow^s_{AN} a_1$.*

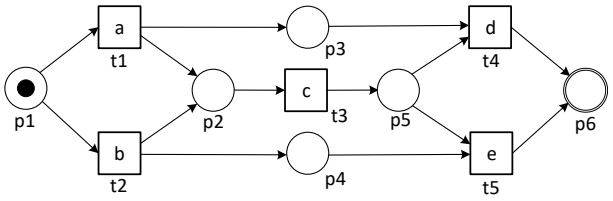


Fig. 5. Accepting Petri net AN_3 that is *not* structural directly-follows complete.

$act^s(AN) = act^b(AN)$ by definition for sound accepting fully labeled Petri nets. However, $df^s(AN)$ and $df^b(AN)$ do not need to be the same. Consider for example accepting Petri net AN_3 in Figure 5. $df^s(AN_3) = \{(\blacktriangleright, a), (\blacktriangleright, b), (a, c), (a, d), (a, e), (b, c), (b, d), (b, e), (c, d), (c, e), (d, \blacksquare), (e, \blacksquare)\}$, but $df^b(AN_3) = \{(\blacktriangleright, a), (\blacktriangleright, b), (a, c), (b, c), (c, d), (c, e), (d, \blacksquare), (e, \blacksquare)\}$, i.e., the relations corresponding to places $p3$ and $p4$ are missing.

Based on accepting Petri net AN_3 in Figure 5, we consider two related event logs: $L_3 = [\langle a, c, d\rangle^{10}, \langle b, c, e\rangle^{10}]$ and $L'_3 = [\langle a, c, d\rangle^5, \langle a, c, e\rangle^5, \langle b, c, d\rangle^5, \langle b, c, e\rangle^5]$. Event log L_3 could have been generated by simulating AN_3, but L'_3 could not (e.g., $\langle a, c, e\rangle$ is not possible). However, both have the same directly-follows relations (shown in Figure 6). Both event logs are directly-follows complete for AN_3.

Definition 15 (Directly-Follows Complete Log). *Let $L \in \mathcal{B}(\mathcal{U}_{act}{}^*)$ be an event log and $AN = (N, M_{init}, M_{final}) \in \mathcal{U}_{AN}$ be an accepting Petri net. L is directly-follows complete for AN if $df(L) = df^b(AN)$.*

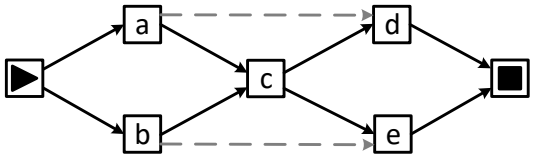


Fig. 6. Directly-follows graph based on the behavior of the accepting Petri net in Figure 5. Note that the long-term dependencies between a and d, and b and e are missing because these activities never directly follow one another.

Note that $df(L_3) = df(L_3') = df^b(AN_3) = \{(\blacktriangleright, a), (\blacktriangleright, b), (a, c), (b, c), (c, d), (c, e), (d, \blacksquare), (e, \blacksquare)\}$. Hence, both event logs are directly-follows complete with respect to AN_3.

Obviously, models like AN_3 are difficult to discover. Region-based techniques are able to find the places $p3$ and $p4$, but are unusable for real-life data sets because they produce complex overfitting process models and are not scalable. Therefore, we are interested in the class of *structural directly-follows complete* accepting Petri nets.

Definition 16 (Structural Directly-Follows Complete Model). *Let $AN = (N, M_{init}, M_{final}) \in \mathcal{U}_{AN}$ be a sound accepting Petri net with $N = (P, T, F, l)$ and $dom(l) = T$. AN is structural directly-follows complete if $df^s(AN) = df^b(AN)$.*

Note that Definition 16 does not depend on a log. It is a model property. AN_1 in Figure 1 is structural directly-follows complete, AN_3 in Figure 5 is not. Figure 7 shows that the free-choice property and structural directly-follows completeness are independent.

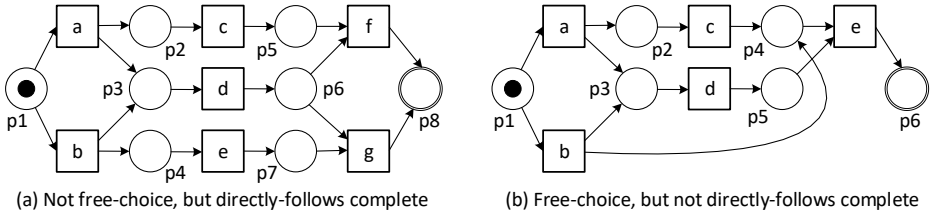


Fig. 7. Two accepting Petri net: (a) AN_4 is *not* free-choice, but structural directly-follows complete (e.g., d can be directly followed by both f and g) and (b) AN_5 is free-choice, but *not* structural directly-follows complete (despite $p4$, activity b is never directly followed by e).

4 Subclasses of Accepting Petri Nets Relevant For Process Mining

Properties such as soundness, liveness, and boundedness are behavioral properties. Structural directly-follows completeness is both structural and behavioral. In this section, we list properties relevant for process mining that are structural.

Definition 17 (Structural Properties). *Let $N = (P, T, F, l)$ be Petri net.*

- *N is a state machine if for all $t \in T$: $|{\bullet t}| \leq 1$ and $|t{\bullet}| \leq 1$.*
- *N is a marked graph if for all $p \in P$: $|{\bullet p}| \leq 1$ and $|p{\bullet}| \leq 1$.*
- *N is free-choice if for all $t_1, t_2 \in T$ with ${\bullet t_1} \cap {\bullet t_2} \neq \emptyset$: ${\bullet t_1} = {\bullet t_2}$.*
- *N is uniwired if for all $t_1, t_2 \in T$: $|t_1{\bullet} \cap {\bullet t_2}| \leq 1$.*
- *N is join-free if for all $p \in P$ and $t \in p{\bullet}$: $|{\bullet p}| \leq 1$ or $|{\bullet t}| \leq 1$.*
- *N is free of self-loops if for all $t \in T$: ${\bullet t} \cap t{\bullet} \neq \emptyset$.*
- *N is free of length-two loops if for all $t_1, t_2 \in T$: $t_1{\bullet} \cap {\bullet t_2} = \emptyset$ or $t_2{\bullet} \cap {\bullet t_1} = \emptyset$.*
- *N is free of PT handles if for all $p \in P$ and $t \in T$ there are no two elementary paths from p to t sharing only p and t.*
- *N is free of TP handles if for all $p \in P$ and $t \in T$ there are no two elementary paths from t to p sharing only p and t.*
- *N is a workflow net if there is one source place $p_{\blacktriangleright} \in P$ and one sink place $p_{\blacksquare} \in P$ such that ${\bullet p_{\blacktriangleright}} = p_{\blacksquare}{\bullet} = \emptyset$ and all nodes are on a path from $p_{\blacktriangleright}$ to $p_{\blacksquare}$.*

Note that these properties consider only the net structure (i.e., the initial marking and behavior are not considered). Free-choice nets separate choice and synchronization and most process discovery algorithms aim to produce free-choice nets (note that process trees and basic BPMN models with XOR and AND gateways correspond to free-choice nets). Block-structured process models (e.g., process trees) are also free of PT and TP handles. The basic Alpha algorithm has difficulties dealing with self-loops, length-two loops, and non-free-choice constructs. Workflow nets have places explicitly indicating the start and the end of the process.

In the remainder, we focus on *regular accepting Petri nets* as our target model. Such nets have desirable properties such as safeness, soundness, and have no silent or duplicate activities.

Definition 18 (Regular Accepting Petri Nets). *$AN = (N, M_{init}, M_{final}) \in \mathcal{U}_{AN}$ is a regular accepting Petri net if AN is safe, sound, and l is bijective (i.e., a one-to-one correspondence between transitions and activities).*

The following theorem is a generalization of Theorem 4.1 in [5].

Theorem 1. *Let $AN = (N, M_{init}, M_{final}) \in \mathcal{U}_{AN}$ be a regular accepting Petri net. For any two transitions $t_1, t_2 \in T$ such that $t_1 \bullet \cap \bullet\, t_2 = \emptyset$: if $l(t_1) \Rightarrow^b_{AN} l(t_2)$, then $l(t_2) \Rightarrow^b_{AN} l(t_1)$.*

Proof. For simplicity, we assume that l is the identity function (this can be achieved by renaming transitions using bijection l). Let $a, b \in T$, $a \bullet \cap \bullet\, b = \emptyset$, and $a \Rightarrow^b_{AN} b$. We need to prove that $b \Rightarrow^b_{AN} a$. If $a = b$, this holds trivially. Hence, we assume $a \neq b$. Because $a \Rightarrow^b_{AN} b$, there is a trace $\sigma_1 \cdot \langle a, b \rangle \cdot \sigma_2 \in lang(AN)$. Because $a \bullet \cap \bullet\, b = \emptyset$, $\sigma_1 \cdot \langle b \rangle$ is enabled. If $\sigma_1 \cdot \langle b, a \rangle$ is not enabled, then b removes a token from an input place of a without returning it (remember that AN is safe). However, if a consumes the token first, then b can no longer fire (leading to a contradiction), i.e., $\sigma_1 \cdot \langle b, a \rangle$ is enabled. Therefore, $\sigma_1 \cdot \langle b, a \rangle \cdot \sigma_2 \in lang(AN)$ and $b \Rightarrow^b_{AN} a$. □

5 The $\alpha^{1.0}$ and $\alpha^{1.1}$ Algorithms

The original Alpha algorithm was developed over two decades ago [5]. We refer to the original algorithm as $\alpha^{1.0}$. It was described in publications such as [1, 5] and during the development of $\alpha^{1.0}$ it was already proven that any Structured Workflow Net (SWN) without self-loops can be "rediscovered", i.e., a directly-follows complete event log obtained by simulating SWN contains enough information for the algorithm to discover the SWN again (modulo renaming of places). An SWN is free-choice, join-free, has a source and sink place, and no implicit places (cf. Definition 17). As pointed out in [1, 5], $\alpha^{1.0}$ has many known limitations. The algorithm assumes that the event log is directly-follows complete and that all behavior observed should be captured in the model (i.e., no noise and infrequent behavior). Moreover, it has problems dealing with short loops and processes that cannot be expressed as a workflow net.

Over the last two decades, there have been many proposals to "repair" $\alpha^{1.0}$ e.g., [24, 25]. However, these alternative approaches assume information that goes beyond the directly-follows relation and most of them are much more complicated (with many case distinctions). In this paper, we propose two variants of $\alpha^{1.0}$: $\alpha^{1.1}$ and $\alpha^{2.0}$. These are as simple as the original algorithm, but allow for the discovery of a larger class of process models. In this section, we introduce $\alpha^{1.1}$ which is close to $\alpha^{1.0}$ and only changes the initial and final parts of the process model to allow for non-workflow nets.

Definition 19 ($\alpha^{1.1}$ Algorithm). *The $\alpha^{1.1}$ algorithm implements a function $disc_{\alpha^{1.1}} \in \mathcal{B}(\mathcal{U}_{act}{}^*) \rightarrow \mathcal{U}_{AN}$ that returns an accepting Petri net $disc_{\alpha^{1.1}}(L)$ for any event log $L \in \mathcal{B}(\mathcal{U}_{act}{}^*)$. Let $A = act(L)$ and $\hat{A} = A \cup \{\blacktriangleright, \blacksquare\}$.*

1. $Cnd(L) = \{(A_1, A_2) \mid A_1 \subseteq \hat{A} \wedge A_1 \neq \emptyset \wedge A_2 \subseteq \hat{A} \wedge A_2 \neq \emptyset \wedge (\forall_{a_1 \in A_1} \forall_{a_2 \in A_2}\ a_1 \Rightarrow_L a_2 \wedge a_2 \not\Rightarrow_L a_1) \wedge (\forall_{a_1, a_2 \in A_1}\ a_1 \not\Rightarrow_L a_2) \wedge (\forall_{a_1, a_2 \in A_2}\ a_1 \not\Rightarrow_L a_2)\}$ *are the candidate places,*
2. $Sel(L) = \{(A_1, A_2) \in Cnd(L) \mid \forall_{(A_1', A_2') \in Cnd(L)}\ A_1 \subseteq A_1' \wedge A_2 \subseteq A_2' \Longrightarrow (A_1, A_2) = (A_1', A_2')\}$ *are the selected maximal places,*
3. $P = \{p_{(A_1, A_2)} \mid (A_1, A_2) \in Sel(L)\}$ *is the set of all places,*
4. $T = \{t_a \mid a \in A\}$ *is the set of transitions,*
5. $F = \{(t_a, p_{(A_1, A_2)}) \mid (A_1, A_2) \in Sel(L) \wedge a \in A_1 \cap A\} \cup \{(p_{(A_1, A_2)}, t_a) \mid (A_1, A_2) \in Sel(L) \wedge a \in A_2 \cap A\}$ *is the set of arcs,*
6. $l = \{(t_a, a) \mid a \in A\}$ *is the labeling function,*
7. $M_{init} = [p_{(A_1, A_2)} \in P \mid \blacktriangleright \in A_1]$ *is the initial marking,* $M_{final} = [p_{(A_1, A_2)} \in P \mid \blacksquare \in A_2]$ *is the final marking, and*
8. $disc_{\alpha^{1.1}}(L) = ((P, T, F, l), M_{init}, M_{final})$ *is the discovered accepting Petri net.*

The first two steps are most important. Set $Sel(L)$ defines the set of places in terms of preceding and succeeding activities. Steps 3–8 are mostly bookkeeping, i.e., all the elements in $Sel(L)$ are mapped onto places with the corresponding connections. $\blacktriangleright$ and $\blacksquare$ are placeholders for the start and end of the process and are not mapped onto transitions, but define the initial and final marking.

Figure 8 shows a simple example highlighting the difference between $\alpha^{1.0}$ and $\alpha^{1.1}$. For the event log $L = [\langle a, b\rangle^{10}, \langle b, a\rangle^{10}]$, the $\alpha^{1.1}$ algorithm specified in Definition 19 generates the correct regular accepting Petri net shown in Figure 8(a). $\alpha^{1.0}$

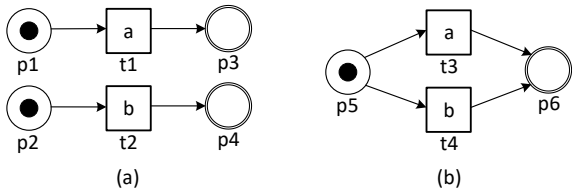


Fig. 8. Improvement over the original algorithm: (a) shows the correct model AN_6 obtained by $\alpha^{1.1}$ and (b) shows the incorrect model AN'_6 obtained by $\alpha^{1.0}$ for the event log $L = [\langle a, b\rangle^{10}, \langle b, a\rangle^{10}]$.

tries to create a workflow net with only one initially marked place (cf. Figure 8(b)). The model in Figure 8(b) is unable to replay any of the traces in the event log, whereas the model in Figure 8(a) is able to replay all and does not allow for unseen behavior. The original algorithm does not allow for concurrent initial and final activities and also does not allow for initial and final activities occurring at different positions. For $L = [\langle a, b\rangle^{10}, \langle b, a\rangle^{10}]$: $\blacktriangleright \Rightarrow_L a$, $\blacktriangleright \Rightarrow_L b$, $a \Rightarrow_L b$, $b \Rightarrow_L a$, $a \Rightarrow_L \blacksquare$, and $b \Rightarrow_L \blacksquare$. Hence, $Cnd(L) = Sel(L) = \{(\{\blacktriangleright\}, \{a\}), (\{\blacktriangleright\}, \{b\}), (\{a\}, \{\blacksquare\}), (\{b\}, \{\blacksquare\})\}$. Note that $p1$ corresponds to $(\{\blacktriangleright\}, \{a\})$, etc.

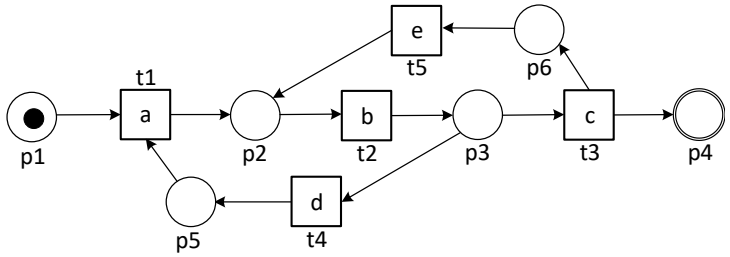


Fig. 9. $\alpha^{1.0}$ produces an incorrect model for $L_4 = [\langle a, b, c\rangle^{10}, \langle a, b, d, a, b, c\rangle^5, \langle a, b, c, e, b, c\rangle^5, \langle a, b, d, a, b, c, e, b, c\rangle^2, \langle a, b, c, e, b, d, a, b, c\rangle^2]$. Note that none of the traces can be replayed.

Another event log where $\alpha^{1.0}$ fails and $\alpha^{1.1}$ produces the desired model is $L_4 = [\langle a, b, c\rangle^{10}, \langle a, b, d, a, b, c\rangle^5, \langle a, b, c, e, b, c\rangle^5, \langle a, b, d, a, b, c, e, b, c\rangle^2, \langle a, b, c, e, b, d, a, b, c\rangle^2]$ for which $\blacktriangleright \Rightarrow_L a$, $a \Rightarrow_L b$, $b \Rightarrow_L c$, $b \Rightarrow_L d$, $c \Rightarrow_L e$, $c \Rightarrow_L \blacksquare$, $d \Rightarrow_L a$, and $e \Rightarrow_L b$. $Sel(L) = \{(\{\blacktriangleright, d\}, \{a\}), (\{a, e\}, \{b\}), (\{b\}, \{c, d\}), (\{c\}, \{e, \blacksquare\})\}$ describes the four places, e.g., $p_{(\{\blacktriangleright, d\}, \{a\})}$ is initially marked, $\bullet p_{(\{\blacktriangleright, d\}, \{a\})} = \{d\}$, and $p_{(\{\blacktriangleright, d\}, \{a\})}\bullet = \{a\}$. Figure 9 shows the incorrect model produced by the $\alpha^{1.0}$ algorithm. Figure 10 shows the correct model produced by the $\alpha^{1.1}$ (and later $\alpha^{2.0}$) algorithm. Note that the latter model is able to replay all traces.

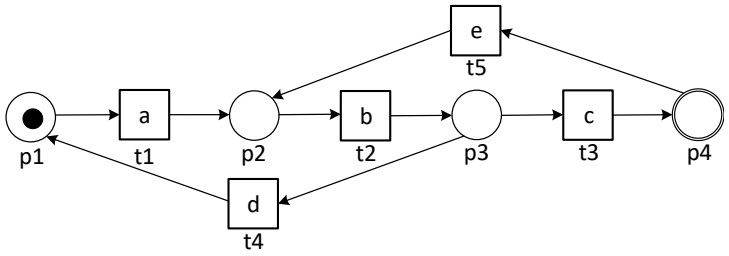


Fig. 10. $\alpha^{1.1}$ produces the desired model is L_4.

6 The $\alpha^{2.0}$ Algorithm

The $\alpha^{1.1}$ algorithm overcomes some of the limitations of the original $\alpha^{1.0}$ algorithm, but for "internal places" the characteristics are essentially the same and short-loops are still a problem. For $L_1 = [\langle a, c, e, f\rangle^5, \ldots]$ in Figure 1, still the incorrect accepting Petri net AN'_1 (Figure 2) is discovered due to the self-loop involving b and the length-two loop involving c and d. The requirements used in $Cnd(L)$ (Definition 19) are always violated for short loops. For any $(A_1, A_2) \in Cnd(L)$: if $a \Rightarrow_L a$, then a cannot be part of A_1 and a cannot be part of A_2. For any loop of length 2 involving a and b, we have $a \Rightarrow_L b$ and $b \Rightarrow_L a$ and the condition $\forall_{a_1 \in A_1} \forall_{a_2 \in A_2}\ a_1 \Rightarrow_L a_2\ \wedge\ a_2 \not\Rightarrow_L a_1$ used in the computation of $Cnd(L)$ is violated for places connecting a and b.

For L_1, we have $b \Rightarrow_{L_1} b$ due to the self-loop involving b, and $c \Rightarrow_{L_1} d$ and $d \Rightarrow_{L_1} c$ due to the length-two loop involving c and d. Therefore, there cannot be a place connecting b to itself or a place connecting c to d or d to c. This can be solved by changing only the first step of the $\alpha^{1.1}$ algorithm specified in Definition 19.

Definition 20 ($\alpha^{2.0}$ Algorithm). *The $\alpha^{2.0}$ algorithm implements a function $disc_{\alpha^{2.0}} \in \mathcal{B}(\mathcal{U}_{act}{}^*) \rightarrow \mathcal{U}_{AN}$ that only differs from $disc_{\alpha^{1.1}}$ in the first step (computation of $Cnd(L)$). The rest is the same as in Definition 19.*

$$
\begin{aligned}
Cnd^{2.0}(L) = \{(A_1, A_2) \mid\ & A_1 \subseteq \hat{A}\ \wedge\ A_2 \subseteq \hat{A}\ \wedge && (1)\\
& (\forall_{a_1 \in A_1} \forall_{a_2 \in A_2}\ a_1 \Rightarrow_L a_2)\ \wedge && (2)\\
& (\exists_{a_1 \in A_1 \setminus A_2} \exists_{a_2 \in A_2 \setminus A_1}\ a_2 \not\Rightarrow_L a_1)\ \wedge && (3)\\
& (\forall_{a_1 \in A_1}\ \forall_{a_2 \in A_1 \setminus A_2}\ a_1 \not\Rightarrow_L a_2)\ \wedge && (4)\\
& (\forall_{a_1 \in A_2 \setminus A_1}\ \forall_{a_2 \in A_2}\ a_1 \not\Rightarrow_L a_2)\} && (5)
\end{aligned}
$$

Candidate places are again represented by pairs of sets of activities. $(A_1, A_2) \in Cnd^{2.0}(L)$ should still be such that elements of A_1 are in a directly-follows relation with elements of A_2. However, it is no longer required that the reverse never holds, i.e., we no longer demand that $\forall_{a_1 \in A_1} \forall_{a_2 \in A_2}\ a_2 \not\Rightarrow_L a_1$. Instead, we assume the weaker requirement that $\exists_{a_1 \in A_1 \setminus A_2} \exists_{a_2 \in A_2 \setminus A_1}\ a_2 \not\Rightarrow_L a_1$. This implies that $A_1 \neq \emptyset$ and $A_2 \neq \emptyset$. Note that $a_1 \in A_1 \setminus A_2$ and $a_2 \in A_2 \setminus A_1$ such that $a_2 \not\Rightarrow_L a_1$ ensures that there is an activity a_1 producing a token for the place without removing it and an activity a_2 consuming a token from the place without putting it back.

Consider the event log $L = [\langle a, b, d\rangle^{10}, \langle a, b, c, b, d\rangle^{3}, \langle a, b, c, b, c, b, d\rangle^{2}, \langle a, b, c, b, c, b, c, b, d\rangle]$. The $\alpha^{2.0}$ algorithm discovers the loop of length two and returns the correct Petri net. The model returned by $\alpha^{1.0}$ and $\alpha^{1.1}$ only allows for trace $\langle a, b, d\rangle$ and leaves c disconnected from the rest.

The $\alpha^{2.0}$ algorithm is also able to discover the regular accepting Petri net AN_1 shown in Figure 1. Note that $\alpha^{2.0}$ inherits all the improvements of the $\alpha^{1.1}$ algorithm. Actually, the algorithm is able to discover all structural directly-follows complete models in this paper. Describing the exact conditions under which the $\alpha^{2.0}$ algorithm is able to rediscover the correct model based on a directly-follows complete event log is outside the scope of the paper. However, the examples clearly show that the class of correctly discovered process models is extended significantly.

The $\alpha^{1.1}$ and $\alpha^{2.0}$ algorithms have been implemented in ProM by Aaron Küsters. (The reader can download the ProM Nightly Build from `www.promtools.org`.) Experiments show that for most event logs more places can be discovered compared to the original algorithm, i.e., there are fewer transitions without any connecting places. Since there is no guarantee that all places are fitting for arbitrary processes, a check has been added to remove places that cannot replay a predefined percentage of cases.

7 Discussion

The $\alpha^{2.0}$ algorithm is able to rediscover accepting Petri nets such as AN_1 in Figure 1, AN_4 in Figure 7(a), AN_6 in Figure 8(a) based on a directly-follows complete event log. The $\alpha^{2.0}$ algorithm is of course *unable to discover models from event logs that are not directly-follows complete*. It fully depends on $\Rightarrow_L$, but this is a reasonable assumption. What has not been observed cannot be discovered! The $\alpha^{2.0}$ algorithm is also *unable to discover models that are not structural directly-follows complete*. The notion of structural directly-follows complete was introduced in this paper. Accepting Petri net AN_3 in Figure 5 is not structural directly-follows complete because $a \not\Rightarrow_L d$ and $b \not\Rightarrow_L e$ although there are places ($p3$ and $p4$) connecting these activities. Accepting Petri net AN_5 in Figure 7(b) is not structural directly-follows complete, because $b \not\Rightarrow_L e$ while there is a place ($p4$) connecting b to e.

Such problems are unavoidable when assuming directly-follows completeness. Therefore, directly-follows-based algorithms like the $\alpha^{2.0}$ algorithm need to be supported by pre- and post-processing techniques. Here, we discuss some examples.

An obvious *preprocessing* step is the filtering of activities and variants as described in [2]. The approach is to first remove infrequent or chaotic activities and then order the remaining variants by frequency (selecting the most frequent ones). Note that most event logs follow a Pareto distribution, i.e., most of the behavior is explained by a limited number of activities and variants.

An obvious *postprocessing* step it to remove places that do not fit. One can use the $\alpha^{2.0}$ algorithm and then check every place individually. This can be done very efficiently. For a place p, one can first look at the sum of the absolute frequencies of the activities represented by $\bullet p$ and compare this with the sum of the absolute frequencies of the activities represented by $p\bullet$. If there is a substantial mismatch, the place can be discarded or repaired. It is also possible to project the event log onto activities $\bullet p \cup p\bullet$

and perform token-based replay or alignments. This can be done very efficiently for a single place. It is possible to set a threshold to retain only the places that fit a minimum percentage of traces. In general, process mining tools should avoid producing models that have known problems.

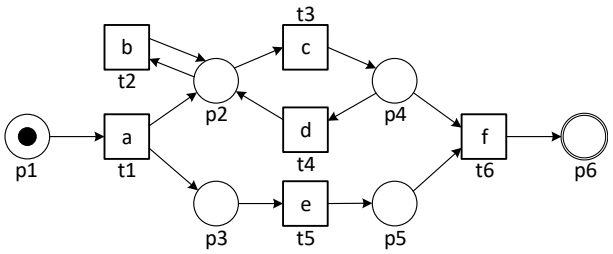


Fig. 11. Accepting Petri net AN'_1 having the same directly-follows relations as AN_1, i.e., $df^b(AN_1) = df^b(AN'_1)$.

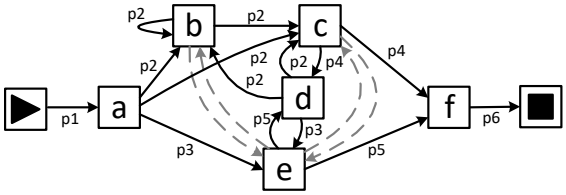
It is also possible to improve discovery by exploiting the fact that we are interested in *safe* accepting Petri nets, i.e., there should not be two tokens in the same place for a particular case. This can be exploited to rule out certain constructs and speed-up the implementation of the algorithm and postprocessing.

It is important to note that there are processes that are behaviorally different, but have the same directly-follows relation. Figure 11 shows an accepting Petri net AN'_1 such that $df^b(AN_1) = df^b(AN'_1)$, i.e., the corresponding DFGs are identical (the DFG shown in Figure 3) although the behaviors are different. This shows the weakness of any algorithm relying on directly-follows relations only.

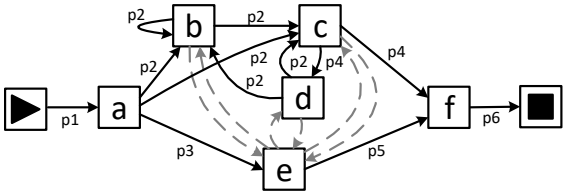
Figure 12(a) maps the arcs of the DFG shown in Figure 3 (based on event log L_1) onto the places of accepting Petri net AN_1 in Figure 1. Figure 12(b) maps the same arcs onto the places of accepting Petri net AN'_1 in Figure 11. Note the difference in the connections between d and e. Figure 12 nicely illustrates that bidirectional arcs either correspond to concurrency or loops of length two. Often it is clear whether bidirectional arcs correspond to concurrency or loops of length two. However, this cannot always be decided based on just the DFG as the example shows.

Another possible refinement is to consider more elements of $Cnd^{2.0}(L)$. We now consider only the maximal elements when applying $Sel(L)$ to the candidates generated in the first step. However, if we can quickly check places on projected event logs, it is possible to select the "largest one" of good quality. Instead of replaying, we can first check the total number of produced and consumed tokens for a place. Given $(A_1, A_2) \in Cnd^{2.0}(L)$, we can compare $\sum_{\sigma \in L} \sum_{a \in A_1} |[a \in \sigma]|$ (tokens produced for the candidate place) and $\sum_{\sigma \in L} \sum_{a \in A_2} |[a \in \sigma]|$ (tokens consumed from the candidate place). If these values are very different, we can discard candidate (A_1, A_2). Instead, we pick the maximal candidate not having obvious quality problems.

Finally, there is the observation that regular accepting Petri nets (cf. Definition 18) *cannot* produce arbitrary directly-follows relations $\Rightarrow^b_{AN}$. There is no regular accepting



(a) Mapping of DFG arcs onto Petri net places in accepting Petri net AN_1



(b) Mapping of DFG arcs onto Petri net places in accepting Petri net AN_1'

Fig. 12. Mapping the Directly-Follows Graph (DFG) based on L_1 onto the places of AN_1 and AN_1'.

Petri net AN with $df^b(AN) = \{(\blacktriangleright, a), (a, b), (a, c), (b, c), (c, \blacksquare)\}$. This knowledge can be used to replace problematic edges in the directly-follows relation by silent activities before applying the $\alpha^{2.0}$ algorithm. In this example, one should replace (a, c) by (a, τ) and (τ, c) and then the $\alpha^{2.0}$ algorithm produces the desired result.

8 Conclusion

Discovering accepting Petri nets from example data in such a way that the resulting model is not overfitting or underfitting is extremely challenging. One cannot assume that all possible behaviors are in the sample log. Therefore, one needs to resort to assumptions like directly-follows completeness. The original Alpha algorithm ($\alpha^{1.0}$) was based on this assumption. In this paper, we tried to push the boundaries of what can be discovered using this assumption and also introduced the new concept of *structural* directly-follows completeness. The insights obtained led to two new variants of the original $\alpha^{1.0}$ algorithm: $\alpha^{1.1}$ and $\alpha^{2.0}$. The $\alpha^{2.0}$ can correctly discover short-loops and non-workflow-net structures. Combining these with the usual pre- and post-processing steps (i.e., filtering and local checks) results in a discovery algorithm that is practically usable. Future work aims at extensive experimentation of the presented approach and detecting/repairing structures in directly-follows relations that cannot stem from a process corresponding to regular accepting Petri net.

Acknowledgment
Funded by the Alexander von Humboldt (AvH) Stiftung and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2023 Internet of Production – 390621612.

References

1. W.M.P. van der Aalst. *Process Mining: Data Science in Action*. Springer-Verlag, Berlin, 2016.
2. W.M.P. van der Aalst. A Practitioner's Guide to Process Mining: Limitations of the Directly-Follows Graph. In *International Conference on Enterprise Information Systems (Centeris 2019)*, volume 164 of *Procedia Computer Science*, pages 321–328. Elsevier, 2019.
3. W.M.P. van der Aalst, A. Adriansyah, and B. van Dongen. Replaying History on Process Models for Conformance Checking and Performance Analysis. *WIREs Data Mining and Knowledge Discovery*, 2(2):182–192, 2012.
4. W.M.P. van der Aalst, V. Rubin, H.M.W. Verbeek, B.F. van Dongen, E. Kindler, and C.W. Günther. Process Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting. *Software and Systems Modeling*, 9(1):87–111, 2010.
5. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
6. F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. Vaandrager. Automata Learning through Counterexample Guided Abstraction Refinement. In *Formal Methods (FM 2012)*, volume 7436 of *Lecture Notes in Computer Science*, pages 10–27. Springer-Verlag, Berlin, 2012.
7. F. Aarts, F. Heidarian, and F. Vaandrager. A Theory of History Dependent Abstractions for Learning Interface Automata. In *CONCUR 2012*, volume 7454 of *Lecture Notes in Computer Science*, pages 240–255. Springer-Verlag, Berlin, 2012.
8. D. Angluin and C.H. Smith. Inductive Inference: Theory and Methods. *Computing Surveys*, 15(3):237–269, 1983.
9. A. Augusto, R. Conforti, M. Marlon, M. La Rosa, and A. Polyvyanyy. Split Miner: Automated Discovery of Accurate and Simple Business Process Models from Event Logs. *Knowledge Information Systems*, 59(2):251–284, 2019.
10. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Process Mining Based on Regions of Languages. In G. Alonso, P. Dadam, and M. Rosemann, editors, *International Conference on Business Process Management (BPM 2007)*, volume 4714 of *Lecture Notes in Computer Science*, pages 375–383. Springer-Verlag, Berlin, 2007.
11. J. Carmona, J. Cortadella, and M. Kishinevsky. A Region-Based Algorithm for Discovering Petri Nets from Event Logs. In *Business Process Management (BPM 2008)*, pages 358–373, 2008.
12. J. Carmona, B. van Dongen, A. Solti, and M. Weidlich. *Conformance Checking: Relating Processes and Models*. Springer-Verlag, Berlin, 2018.
13. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.
14. J. Desel and W. Reisig. Place/Transition Nets. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 122–173. Springer-Verlag, Berlin, 1998.
15. A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures - Part 1 and Part 2. *Acta Informatica*, 27(4):315–368, 1989.
16. M. Kerremans, T. Srivastava, and F.Choudhary. Gartner Market Guide for Process Mining, Research Note G00737056. `www.gartner.com`, 2021.
17. S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. Discovering Block-structured Process Models from Event Logs: A Constructive Approach. In J.M. Colom and J. Desel, editors, *Applications and Theory of Petri Nets 2013*, volume 7927 of *Lecture Notes in Computer Science*, pages 311–329. Springer-Verlag, Berlin, 2013.

18. S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour. In N. Lohmann, M. Song, and P. Wohed, editors, *Business Process Management Workshops, International Workshop on Business Process Intelligence (BPI 2013)*, volume 171 of *Lecture Notes in Business Information Processing*, pages 66–78. Springer-Verlag, Berlin, 2014.
19. S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. Scalable Process Discovery and Conformance Checking. *Software and Systems Modeling*, 17(2):599–631, 2018.
20. L. Mannel and W.M.P. van der Aalst. Finding Complex Process-Structures by Exploiting the Token-Game. In S. Donatelli and S. Haar, editors, *Applications and Theory of Petri Nets 2019*, volume 11522 of *Lecture Notes in Computer Science*, pages 258–278. Springer-Verlag, Berlin, 2019.
21. M. Solé and J. Carmona. Process Mining from a Basis of State Regions. In J. Lilius and W. Penczek, editors, *Applications and Theory of Petri Nets 2010*, volume 6128 of *Lecture Notes in Computer Science*, pages 226–245. Springer-Verlag, Berlin, 2010.
22. A.F. Syring, N. Tax, and W.M.P. van der Aalst. Evaluating Conformance Measures in Process Mining Using Conformance Propositions. In M. Koutny, L. Pomello, and L.M. Kristensen, editors, *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC 14)*, volume 11970 of *Lecture Notes in Computer Science*, pages 192–221. Springer-Verlag, Berlin, 2019.
23. F. Vaandrager. Model Learning. *Communications of the ACM*, 60(2):86–95, 2002.
24. L. Wen, W.M.P. van der Aalst, J. Wang, and J. Sun. Mining Process Models with Non-Free-Choice Constructs. *Data Mining and Knowledge Discovery*, 15(2):145–180, 2007.
25. L. Wen, J. Wang, W.M.P. van der Aalst, B. Huang, and J. Sun. Mining Process Models with Prime Invisible Tasks. *Data and Knowledge Engineering*, 69(10):999–1021, 2010.
26. J.M.E.M. van der Werf, B.F. van Dongen, C.A.J. Hurkens, and A. Serebrenik. Process Discovery using Integer Linear Programming. *Fundamenta Informaticae*, 94:387–412, 2010.
27. S.J. van Zelst, B.F. van Dongen, W.M.P. van der Aalst, and H.M.W Verbeek. Discovering Workflow Nets Using Integer Linear Programming. *Computing*, 100(5):529–556, 2018.