Chapter 6
Process Discovery: An Introduction

Process discovery is one of the most challenging process mining tasks. Based on an event log a process model is constructed thus capturing the behavior seen in the log. This chapter introduces the topic using the rather naïve α-algorithm. This algorithm nicely illustrates some of the general ideas used by many process mining algorithms and helps to understand the notion of process discovery. Moreover, the α-algorithm serves as a stepping stone for discussing challenges related to process discovery.

6.1 Problem Statement

As discussed in Chap. 2, there are three types of process mining: discovery, conformance, and enhancement. Moreover, we identified various perspectives, e.g., the control-flow perspective, the organizational or resource perspective, the data perspective, and the time perspective. In this chapter, we focus on the *discovery* task and the *control-flow* perspective. This combination is often referred to as *process discovery*. The general process discovery problem can be formulated as follows.

Definition 6.1 (General process discovery problem) Let L be an event log as defined in Definition 5.3 or as specified by the XES standard (cf. Sect. 5.3). A *process discovery algorithm* is a function that maps L onto a process model such that the model is "representative" for the behavior seen in the event log. The challenge is to find such an algorithm.

This definition does not specify what kind of process model should be generated, e.g., a BPMN, EPC, YAWL, or Petri net model. Moreover, event logs with potentially many attributes may be used as input. Recall that the XES format allows for storing information related to all perspectives whereas here the focus is on the control-flow perspective. The only requirement is that the behavior is "representative", but it is unclear what this means.

W. van der Aalst, *Process Mining*, DOI 10.1007/978-3-662-49851-4_6

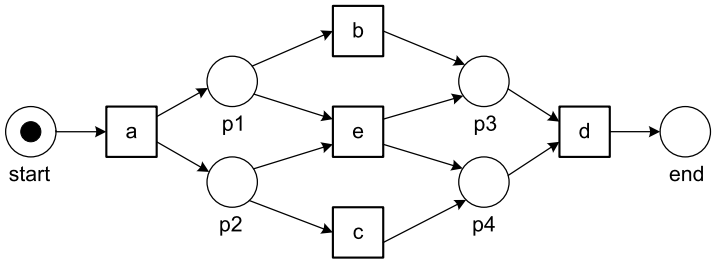


Fig. 6.1 WF-net N_1 discovered for $L_1 = [\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^2, \langle a, e, d\rangle]$

Definition 6.1 is rather broad and vague. The target format is not specified and a potentially "rich" event log is used as input without specifying tangible requirements. To make things more concrete, we define the target to be a Petri net model. Moreover, we use a *simple event log* as input (cf. Definition 5.4). A simple event log L is a multi-set of traces over some set of activities $\mathscr{A}$, i.e., $L \in \mathbb{B}(\mathscr{A}^*)$. For example,

$$L_1 = \left[\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^2, \langle a, e, d\rangle\right]$$

L_1 is a simple log describing the history of six cases. The goal is now to discover a Petri net that can "replay" event log L_1. Ideally, the Petri net is a sound WF-net as defined in Sect. 3.2.3. Based on these choices we reformulate the process discovery problem and make it more concrete.

Definition 6.2 (Specific process discovery problem) A *process discovery algorithm* is a function γ that maps a log $L \in \mathbb{B}(\mathscr{A}^*)$ onto a marked Petri net $\gamma(L) = (N, M)$. Ideally, N is a *sound WF-net* and all traces in L correspond to possible firing sequences of (N, M).

Function γ defines a so-called "Play-In" technique as described in Chap. 2. Based on L_1, a process discovery algorithm γ could discover the WF-net shown in Fig. 6.1, i.e., $\gamma(L_1) = (N_1, [start])$. Each trace in L_1 corresponds to a possible firing sequence of WF-net N_1 shown in Fig. 6.1. Therefore, it is easy to see that the WF-net can indeed replay all traces in the event log. In fact, each of the three possible firing sequences of WF-net N_1 appears in L_1.

Let us now consider another event log,

$$L_2 = \left[\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^4, \langle a, b, c, e, f, b, c, d\rangle^2, \langle a, b, c, e, f, c, b, d\rangle,\right.$$
$$\left.\langle a, c, b, e, f, b, c, d\rangle^2, \langle a, c, b, e, f, b, c, e, f, c, b, d\rangle\right]$$

L_2 is a simple event log consisting of 13 cases represented by 6 different traces. Based on event log L_2, some γ could discover WF-net N_2 shown in Fig. 6.2. This WF-net can indeed replay all traces in the log. However, not all firing sequences of N_2 correspond to traces in L_2. For example, the firing sequence $\langle a, c, b, e, f, c, b, d\rangle$

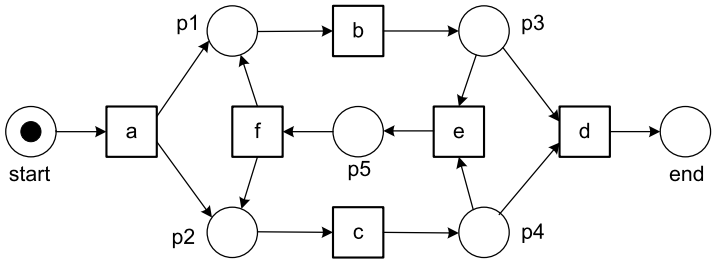


Fig. 6.2 WF-net N_2 discovered for $L_2 = [\langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^4, \langle a,b,c,e,f,b,c,d\rangle^2, \langle a,b,c,e,f,c,b,d\rangle, \langle a,c,b,e,f,b,c,d\rangle^2, \langle a,c,b,e,f,b,c,e,f,c,b,d\rangle]$

does not appear in L_2. In fact, there are infinitely many firing sequences because of the loop construct in N_2. Clearly, these cannot all appear in the event log. Therefore, Definition 6.2 does not require all firing sequences of (N, M) to be traces in L.

In this chapter, we focus on the discovery of Petri nets. The reason is that Petri nets are simple and graphical while still allowing for the modeling of concurrency, choices, and iteration. This is illustrated by Figs. 6.1 and 6.2. In both models activities b and c are concurrent. In N_1, there is choice following a. In N_2, there is choice between d and e each time both b and c complete. Both N_1 and N_2 are sound WF-nets. As explained in Chap. 3, WF-nets are a natural subclass of Petri nets tailored toward the modeling and analysis of operational processes. A process model describes the life-cycle of one case. Therefore, WF-nets explicitly model the creation and the completion of the cases. The creation is modeled by putting a token in the unique source place i (place *start* in Figs. 6.1 and 6.2). The completion is modeled by reaching the state marking the unique sink place o (place *end* in Figs. 6.1 and 6.2). Given a unique source place i and a unique sink place o, the soundness requirement described in Definition 3.7 follows naturally. Recall that a WF-net N is *sound* if and only if

- $(N, [i])$ is *safe*, i.e., places cannot hold multiple tokens at the same time;
- For any marking $M \in [N, [i]\rangle$, $o \in M$ implies $M = [o]$, i.e., if the sink place is marked, all other places should be empty (*proper completion*);
- For any marking $M \in [N, [i]\rangle$, $[o] \in [N, M\rangle$, i.e., it is always possible to mark the sink place (*option to complete*); and
- $(N, [i])$ contains *no dead transitions*, i.e., all parts of the model are potentially reachable.

Most process modeling notations use or assume correctness criteria similar to soundness. For instance, deadlocks and livelocks are symptoms of a process that cannot complete (properly). These phenomena are undesired, independent of the notation used.

Although we use WF-nets in this chapter, this does not imply that discovered process models cannot be presented using other notations. As discussed in Chap. 3, there exist many translations from Petri nets into other notations and vice versa.

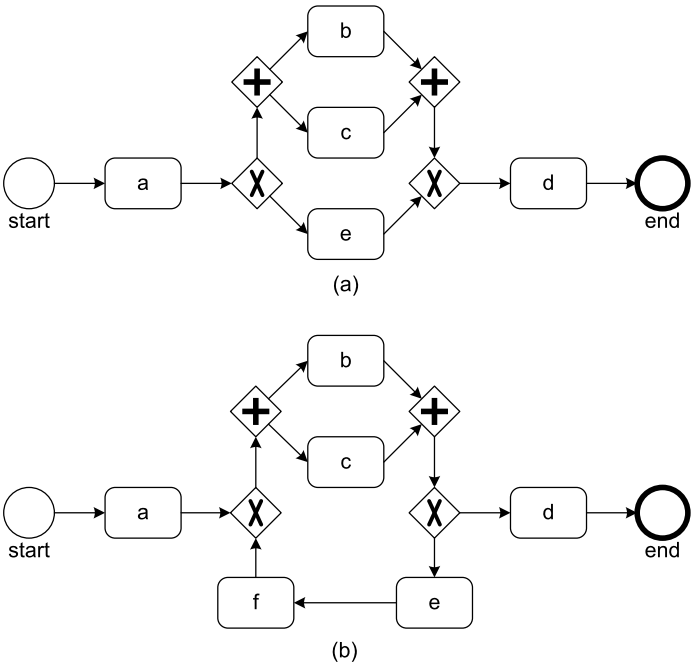


Fig. 6.3 Two BPMN models: (**a**) the model corresponding to WF-net N_1 discovered for L_1, and (**b**) the model corresponding to WF-net N_2 discovered for L_2

Compact formalisms with formal semantics like Petri nets are most suitable to develop and explain process mining algorithms. The representation used to show results to end users is less relevant for the actual process discovery task. For example, the WF-nets depicted in Figs. 6.1 and 6.2 can also be presented in terms of the two trace equivalent BPMN models shown in Fig. 6.3. Similarly, the discovered models could have been translated into equivalent EPCs, UML activity diagrams, statecharts, YAWL models, BPEL specifications, etc.

In the general problem formulation (Definition 6.1) we stated that the discovered model should be "representative" for the behavior seen in the event log. In Definition 6.2, this was operationalized by requiring that the model is able to replay all behavior in this log, i.e., any trace in the event log is a possible firing sequence of the WF-net. This is the so-called "fitness" requirement. In general, there is a trade-off between the following four quality criteria:

- (*Fitness*) The discovered model should allow for the behavior seen in the event log.
- (*Precision*) The discovered model should not allow for behavior completely unrelated to what was seen in the event log.
- (*Generalization*) The discovered model should generalize the example behavior seen in the event log.
- *(Simplicity*) The discovered model should be as simple as possible.

A model having a good fitness is able to replay most of the traces in the log. Precision is related to the notion of *underfitting* presented in the context of data mining (see Sect. 4.6.3). A model having a poor precision is underfitting, i.e., it allows for behavior that is very different from what was seen in the event log. Generalization is related to the notion of *overfitting*. An overfitting model does not generalize enough, i.e., it is too specific and too much driven by examples in the event log. The fourth quality criterion is related to Occam's Razor which states that "one should not increase, beyond what is necessary, the number of entities required to explain anything" (see Sect. 4.6.3). Following this principle, we look for the "simplest process model" that can explain what is observed in the event log.

It turns out to be challenging to balance the four quality criteria. For instance, an oversimplified model is likely to have a low fitness or lack of precision. Moreover, there is an obvious trade-off between underfitting and overfitting. We discuss these four quality criteria later in this chapter. However, we first introduce a concrete process discovery algorithm.

6.2 A Simple Algorithm for Process Discovery

This section introduces the α-algorithm [157]. This algorithm is an example of a γ function as mentioned in Definition 6.2, i.e., given a simple event log it produces a Petri net that (hopefully) can replay the log. The α-algorithm was one of the first process discovery algorithms that could adequately deal with concurrency (see Sect. 7.6). However, the α-algorithm should not be seen as a very practical mining technique as it has problems with noise, infrequent/incomplete behavior, and complex routing constructs. Nevertheless, it provides a good introduction into the topic. The α-algorithm is simple and many of its ideas have been embedded in more complex and robust techniques. We will use the algorithm as a baseline for discussing the challenges related to process discovery and for introducing more practical algorithms.

6.2.1 Basic Idea

Input for the α-algorithm is a simple event log L over $\mathscr{A}$, i.e., $L \in \mathbb{B}(\mathscr{A}^*)$. In the remainder, we will simply refer to L as the event log. We refer to the elements of $\mathscr{A}$ as *activities*, see Sect. 3.2. These activities will correspond to transitions in the discovered Petri net. In this chapter, we will use the convention that capital letters refer to sets of activities (e.g., $A, B \subseteq \mathscr{A}$), whereas for individual activities no capitalization is used (e.g., $a, b, c, \ldots \in \mathscr{A}$). The output of the α-algorithm is a marked Petri net, i.e., $\alpha(L) = (N, M)$. We aim at the discovery of WF-nets. Therefore, we can omit the initial marking and write $\alpha(L) = N$ (the initial marking is implied; $M = [i]$).

Table 6.1 Footprint of L_1: $a\#_{L_1}a$, $a \rightarrow_{L_1} b$, $a \rightarrow_{L_1} c$, etc.

	a	b	c	d	e
a	$\#_{L_1}$	$\rightarrow_{L_1}$	$\rightarrow_{L_1}$	$\#_{L_1}$	$\rightarrow_{L_1}$
b	$\leftarrow_{L_1}$	$\#_{L_1}$	$\|_{L_1}$	$\rightarrow_{L_1}$	$\#_{L_1}$
c	$\leftarrow_{L_1}$	$\|_{L_1}$	$\#_{L_1}$	$\rightarrow_{L_1}$	$\#_{L_1}$
d	$\#_{L_1}$	$\leftarrow_{L_1}$	$\leftarrow_{L_1}$	$\#_{L_1}$	$\leftarrow_{L_1}$
e	$\leftarrow_{L_1}$	$\#_{L_1}$	$\#_{L_1}$	$\rightarrow_{L_1}$	$\#_{L_1}$

The α-algorithm *scans the event log for particular patterns*. For example, if activity a is followed by b but b is never followed by a, then it is assumed that there is a causal dependency between a and b. To reflect this dependency, the corresponding Petri net should have a place connecting a to b. We distinguish four *log-based ordering relations* that aim to capture relevant patterns in the log.

Definition 6.3 (Log-based ordering relations) Let L be an event log over $\mathscr{A}$, i.e., $L \in \mathbb{B}(\mathscr{A}^*)$. Let $a, b \in \mathscr{A}$.

- $a >_L b$ if and only if there is a trace $\sigma = \langle t_1, t_2, t_3, \ldots, t_n \rangle$ and $i \in \{1, \ldots, n-1\}$ such that $\sigma \in L$ and $t_i = a$ and $t_{i+1} = b$;
- $a \rightarrow_L b$ if and only if $a >_L b$ and $b \not>_L a$;
- $a \#_L b$ if and only if $a \not>_L b$ and $b \not>_L a$; and
- $a \|_L b$ if and only if $a >_L b$ and $b >_L a$.

Consider, for instance, $L_1 = [\langle a, b, c, d \rangle^3, \langle a, c, b, d \rangle^2, \langle a, e, d \rangle]$ again. For this event log, the following log-based ordering relations can be found:

$$>_{L_1} = \{(a,b), (a,c), (a,e), (b,c), (c,b), (b,d), (c,d), (e,d)\}$$

$$\rightarrow_{L_1} = \{(a,b), (a,c), (a,e), (b,d), (c,d), (e,d)\}$$

$$\#_{L_1} = \{(a,a), (a,d), (b,b), (b,e), (c,c), (c,e), (d,a), (d,d), (e,b), (e,c), (e,e)\}$$

$$\|_{L_1} = \{(b,c), (c,b)\}$$

Relation $>_{L_1}$ contains all pairs of activities in a "directly follows" relation. $c >_{L_1} d$ because d directly follows c in trace $\langle a, b, c, d \rangle$. However, $d \not>_{L_1} c$ because c never directly follows d in any trace in the log. $\rightarrow_{L_1}$ contains all pairs of activities in a "causality" relation, e.g., $c \rightarrow_{L_1} d$ because sometimes d directly follows c and never the other way around ($c >_{L_1} d$ and $d \not>_{L_1} c$). $b \|_{L_1} c$ because $b >_{L_1} c$ and $c >_{L_1} b$, i.e., sometimes c follows b and sometimes the other way around. $b \#_{L_1} e$ because $b \not>_{L_1} e$ and $e \not>_{L_1} b$.

For any log L over $\mathscr{A}$ and $x, y \in \mathscr{A}$, $x \rightarrow_L y$, $y \rightarrow_L x$, $x \#_L y$, or $x \|_L y$, i.e., precisely one of these relations holds for any pair of activities. Therefore, the *footprint* of a log can be captured in a matrix as shown in Table 6.1.

The footprint of event log L_2 is shown in Table 6.2. The subscripts have been removed to not clutter the table. When comparing the footprints of L_1 and L_2 one can see that only the e and f columns and rows differ.

Table 6.2 Footprint of $L_2 = [\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^4, \langle a, b, c, e, f, b, c, d\rangle^2, \langle a, b, c, e, f, c, b, d\rangle, \langle a, c, b, e, f, b, c, d\rangle^2, \langle a, c, b, e, f, b, c, e, f, c, b, d\rangle]$

	a	b	c	d	e	f
a	#	$\rightarrow$	$\rightarrow$	#	#	#
b	$\leftarrow$	#	$\parallel$	$\rightarrow$	$\rightarrow$	$\leftarrow$
c	$\leftarrow$	$\parallel$	#	$\rightarrow$	$\rightarrow$	$\leftarrow$
d	#	$\leftarrow$	$\leftarrow$	#	#	#
e	#	$\leftarrow$	$\leftarrow$	#	#	$\rightarrow$
f	#	$\rightarrow$	$\rightarrow$	#	$\leftarrow$	#

The log-based ordering relations can be used to *discover patterns* in the corresponding process model as is illustrated in Fig. 6.4. If a and b are in sequence, the log will show $a \rightarrow_L b$. If after a there is a choice between b and c, the log will show $a \rightarrow_L b$, $a \rightarrow_L c$, and $b \#_L c$ because a can be followed by b and c, but b will not be followed by c and vice versa. The logical counterpart of this so-called XOR-split pattern is the XOR-join pattern as shown in Fig. 6.4(b)–(c). If $a \rightarrow_L c$, $b \rightarrow_L c$, and $a \#_L b$, then this suggests that after the occurrence of either a or b, c should happen. Figure 6.4(d)–(e) shows the so-called AND-split and AND-join patterns. If $a \rightarrow_L b$, $a \rightarrow_L c$, and $b \parallel_L c$, then it appears that after a both b and c can be executed in parallel (AND-split pattern). If $a \rightarrow_L c$, $b \rightarrow_L c$, and $a \parallel_L b$, then the log suggests that c needs to synchronize a and b (AND-join pattern).

Figure 6.4 only shows simple patterns and does not present the additional conditions needed to extract the patterns. However, the figure nicely illustrates the basic idea.

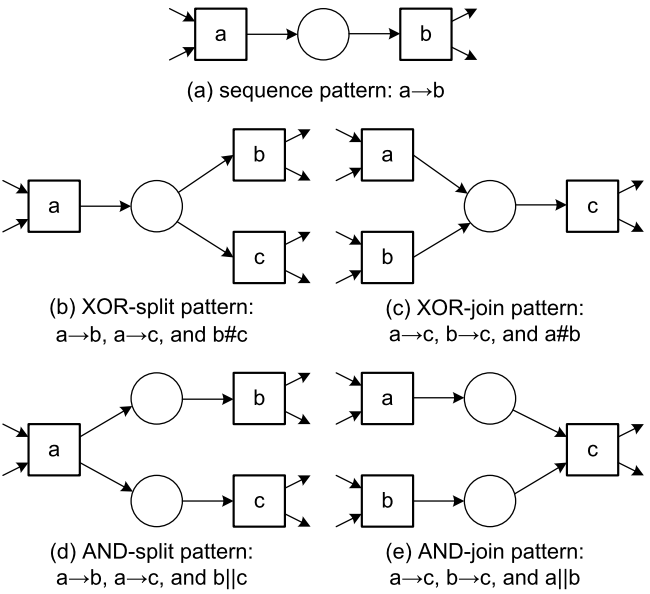


Fig. 6.4 Typical process patterns and the footprints they leave in the event log

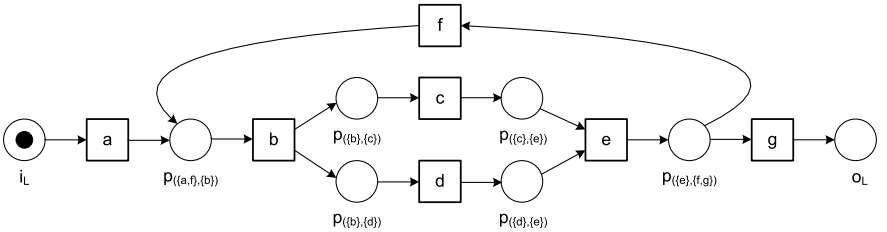


Fig. 6.5 WF-net N_3 derived from $L_3 = [\langle a, b, c, d, e, f, b, d, c, e, g\rangle, \langle a, b, d, c, e, g\rangle^2, \langle a, b, c, d, e, f, b, c, d, e, f, b, d, c, e, g\rangle]$

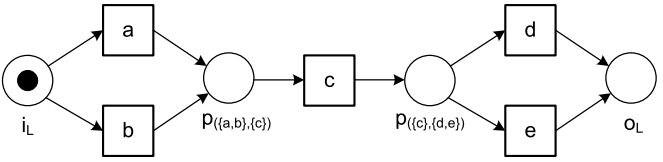


Fig. 6.6 WF-net N_4 derived from $L_4 = [\langle a, c, d\rangle^{45}, \langle b, c, d\rangle^{42}, \langle a, c, e\rangle^{38}, \langle b, c, e\rangle^{22}]$

Table 6.3 Footprint of L_3

	a	b	c	d	e	f	g
a	#	$\rightarrow$	#	#	#	#	#
b	$\leftarrow$	#	$\rightarrow$	$\rightarrow$	#	$\leftarrow$	#
c	#	$\leftarrow$	#	$\parallel$	$\rightarrow$	#	#
d	#	$\leftarrow$	$\parallel$	#	$\rightarrow$	#	#
e	#	#	$\leftarrow$	$\leftarrow$	#	$\rightarrow$	$\rightarrow$
f	#	$\rightarrow$	#	#	$\leftarrow$	#	#
g	#	#	#	#	$\leftarrow$	#	#

Consider, for example, WF-net N_3 depicted in Fig. 6.5 and the event log L_3 describing four cases,

$$L_3 = \big[\langle a, b, c, d, e, f, b, d, c, e, g\rangle, \langle a, b, d, c, e, g\rangle^2, \langle a, b, c, d, e, f, b, c, d, e, f, b, d, c, e, g\rangle\big]$$

The α-algorithm constructs WF-net N_3 based on L_3 (see Fig. 6.5).

Table 6.3 shows the footprint of L_3. Note that the patterns in the model indeed match the log-based ordering relations extracted from the event log. Consider, for example, the process fragment involving b, c, d, and e. Obviously, this fragment can be constructed based on $b \rightarrow_{L_3} c$, $b \rightarrow_{L_3} d$, $c \parallel_{L_3} d$, $c \rightarrow_{L_3} e$, and $d \rightarrow_{L_3} e$. The choice following e is revealed by $e \rightarrow_{L_3} f$, $e \rightarrow_{L_3} g$, and $f \#_{L_3} g$; etc.

Another example is shown in Fig. 6.6. WF-net N_4 can be derived from L_4,

$$L_4 = \left[\langle a, c, d\rangle^{45}, \langle b, c, d\rangle^{42}, \langle a, c, e\rangle^{38}, \langle b, c, e\rangle^{22}\right]$$

L_4 contains information about 147 cases that follow one of the four possible traces. There are two start and two end activities. These can be detected easily by looking for the first and last activities in traces.

6.2.2 Algorithm

After showing the basic idea and some examples, we describe the α-algorithm [157].

Definition 6.4 (α-algorithm) Let L be an event log over $T \subseteq \mathscr{A}$. $\alpha(L)$ is defined as follows:

1. $T_L = \{t \in T \mid \exists_{\sigma \in L}\ t \in \sigma\}$,
2. $T_I = \{t \in T \mid \exists_{\sigma \in L}\ t = \mathit{first}(\sigma)\}$,
3. $T_O = \{t \in T \mid \exists_{\sigma \in L}\ t = \mathit{last}(\sigma)\}$,
4. $X_L = \{(A, B) \mid A \subseteq T_L \wedge A \neq \emptyset \wedge B \subseteq T_L \wedge B \neq \emptyset \wedge \forall_{a \in A} \forall_{b \in B}\ a \rightarrow_L b \wedge \forall_{a_1, a_2 \in A}\ a_1 \#_L a_2 \wedge \forall_{b_1, b_2 \in B}\ b_1 \#_L b_2\}$,
5. $Y_L = \{(A, B) \in X_L \mid \forall_{(A', B') \in X_L} A \subseteq A' \wedge B \subseteq B' \Longrightarrow (A, B) = (A', B')\}$,
6. $P_L = \{p_{(A,B)} \mid (A, B) \in Y_L\} \cup \{i_L, o_L\}$,
7. $F_L = \{(a, p_{(A,B)}) \mid (A, B) \in Y_L \wedge a \in A\} \cup \{(p_{(A,B)}, b) \mid (A, B) \in Y_L \wedge b \in B\} \cup \{(i_L, t) \mid t \in T_I\} \cup \{(t, o_L) \mid t \in T_O\}$, and
8. $\alpha(L) = (P_L, T_L, F_L)$.

L is an event log over some set T of activities. In Step 1, it is checked which activities do appear in the log (T_L). These will correspond to the transitions of the generated WF-net. T_I is the set of start activities, i.e., all activities that appear first in some trace (Step 2). T_O is the set of end activities, i.e., all activities that appear last in some trace (Step 3). Steps 4 and 5 form the core of the α-algorithm. The challenge is to determine the places of the WF-net and their connections. We aim at constructing places named $p_{(A,B)}$ such that A is the set of input transitions ($\bullet p_{(A,B)} = A$) and B is the set of output transitions ($p_{(A,B)}\bullet = B$) of $p_{(A,B)}$.

The basic motivation for finding $p_{(A,B)}$ is illustrated by Fig. 6.7. All elements of A should have causal dependencies with all elements of B, i.e., for all $(a, b) \in A \times B$: $a \rightarrow_L b$. Moreover, the elements of A should never follow one another, i.e., for all $a_1, a_2 \in A$: $a_1 \#_L a_2$. A similar requirement holds for B.

Table 6.4 shows the structure in terms of the footprint matrix introduced earlier. If we *only* consider the columns and rows related to $A \cup B$ and group the rows and columns belonging to A respectively B, we get the pattern shown in Table 6.4. There are four quadrants. Two quadrants only contain the symbol #. This shows that the elements of A should never follow another (upper-left quadrant) and that the elements of B should never follow another (lower-right quadrant). The upper-right

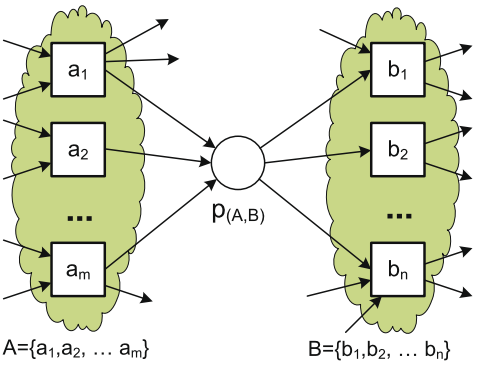


Fig. 6.7 Place $p_{(A,B)}$ connects the transitions in set A to the transitions in set B

Table 6.4 How to identify $(A, B) \in X_L$? Rearrange the rows and columns corresponding to $A = \{a_1, a_2, \ldots, a_m\}$ and $B = \{b_1, b_2, \ldots, b_n\}$ and remove the other rows and columns from the footprint

	a_1	a_2	...	a_m	b_1	b_2	...	b_n
a_1	#	#	...	#	$\rightarrow$	$\rightarrow$	...	$\rightarrow$
a_2	#	#	...	#	$\rightarrow$	$\rightarrow$	...	$\rightarrow$
...	...	...	...	...	...	...	...	...
a_m	#	#	...	#	$\rightarrow$	$\rightarrow$	...	$\rightarrow$
b_1	$\leftarrow$	$\leftarrow$	...	$\leftarrow$	#	#	...	#
b_2	$\leftarrow$	$\leftarrow$	...	$\leftarrow$	#	#	...	#
...	...	...	...	...	...	...	...	...
b_n	$\leftarrow$	$\leftarrow$	...	$\leftarrow$	#	#	...	#

quadrant only contains the symbol $\rightarrow$, any of the elements in A can be followed by any of the elements in B but never the other way around. By symmetry, the lower-left quadrant only contains the symbol $\leftarrow$.

Let us consider L_1 again. Clearly, $A = \{a\}$ and $B = \{b, e\}$ meet the requirements stated in Step 4. Also $A' = \{a\}$ and $B' = \{b\}$ meet the same requirements. X_L is the set of all such pairs that meet the requirements just mentioned. In this case,

$$X_{L_1} = \big\{\big(\{a\}, \{b\}\big), \big(\{a\}, \{c\}\big), \big(\{a\}, \{e\}\big), \big(\{a\}, \{b, e\}\big), \big(\{a\}, \{c, e\}\big),$$
$$\big(\{b\}, \{d\}\big), \big(\{c\}, \{d\}\big), \big(\{e\}, \{d\}\big), \big(\{b, e\}, \{d\}\big), \big(\{c, e\}, \{d\}\big)\big\}$$

If one would insert a place for any element in X_{L_1}, there would be too many places. Therefore, only the "maximal pairs" (A, B) should be included. Note that for any pair $(A, B) \in X_L$, non-empty set $A' \subseteq A$, and non-empty set $B' \subseteq B$, it is implied that $(A', B') \in X_L$. In Step 5, all non-maximal pairs are removed, thus yielding

$$Y_{L_1} = \big\{\big(\{a\}, \{b, e\}\big), \big(\{a\}, \{c, e\}\big), \big(\{b, e\}, \{d\}\big), \big(\{c, e\}, \{d\}\big)\big\}$$

Step 5 can also be understood in terms the footprint matrix. Consider Table 6.4 and let A' and B' be such that $\emptyset \subset A' \subseteq A$ and $\emptyset \subset B' \subseteq B$. Removing rows and columns $A \cup B \setminus (A' \cup B')$ results in a matrix still having the pattern shown in Table 6.4. Therefore, we only consider maximal matrices for constructing Y_L.

Table 6.5 Footprint of L_5

	a	b	c	d	e	f
a	#	$\rightarrow$	#	#	$\rightarrow$	#
b	$\leftarrow$	#	$\rightarrow$	$\leftarrow$	$\parallel$	$\rightarrow$
c	#	$\leftarrow$	#	$\rightarrow$	$\parallel$	#
d	#	$\rightarrow$	$\leftarrow$	#	$\parallel$	#
e	$\leftarrow$	$\parallel$	$\parallel$	$\parallel$	#	$\rightarrow$
f	#	$\leftarrow$	#	#	$\leftarrow$	#

Every element of $(A, B) \in Y_L$ corresponds to a place $p_{(A,B)}$ connecting transitions A to transitions B. In addition P_L also contains a unique source place i_L and a unique sink place o_L (cf. Step 6). Remember that the goal is to create a WF-net.[1]

In Step 7, the arcs of the WF-net are generated. All start transitions in T_I have i_L as an input place and all end transitions T_O have o_L as output place. All places $p_{(A,B)}$ have A as input nodes and B as output nodes. The result is a Petri net $\alpha(L) = (P_L, T_L, F_L)$ that describes the behavior seen in event log L.

Thus far we presented four logs and four WF-nets. Application of the α-algorithm shows that indeed $\alpha(L_3) = N_3$ and $\alpha(L_4) = N_4$. In Figs. 6.5 and 6.6, the places are named based on the sets Y_{L_3} and Y_{L_4}. Moreover, $\alpha(L_1) = N_1$ and $\alpha(L_2) = N_2$ modulo renaming of places (because different place names are used in Figs. 6.1 and 6.2). These examples show that the α-algorithm is indeed able to discover WF-nets based on event logs.

Let us now consider event log L_5,

$$L_5 = \big[\langle a, b, e, f\rangle^2, \langle a, b, e, c, d, b, f\rangle^3, \langle a, b, c, e, d, b, f\rangle^2, \\ \langle a, b, c, d, e, b, f\rangle^4, \langle a, e, b, c, d, b, f\rangle^3\big]$$

Table 6.5 shows the footprint of the log.

Let us now apply the 8 steps of the algorithm for $L = L_5$:

$$T_L = \{a, b, c, d, e, f\}$$
$$T_I = \{a\}$$
$$T_O = \{f\}$$
$$X_L = \big\{\big(\{a\}, \{b\}\big), \big(\{a\}, \{e\}\big), \big(\{b\}, \{c\}\big), \big(\{b\}, \{f\}\big), \big(\{c\}, \{d\}\big), \\ \big(\{d\}, \{b\}\big), \big(\{e\}, \{f\}\big), \big(\{a, d\}, \{b\}\big), \big(\{b\}, \{c, f\}\big)\big\}$$
$$Y_L = \big\{\big(\{a\}, \{e\}\big), \big(\{c\}, \{d\}\big), \big(\{e\}, \{f\}\big), \big(\{a, d\}, \{b\}\big), \big(\{b\}, \{c, f\}\big)\big\}$$
$$P_L = \{p_{(\{a\},\{e\})}, p_{(\{c\},\{d\})}, p_{(\{e\},\{f\})}, p_{(\{a,d\},\{b\})}, p_{(\{b\},\{c,f\})}, i_L, o_L\}$$
$$F_L = \big\{(a, p_{(\{a\},\{e\})}), (p_{(\{a\},\{e\})}, e), (c, p_{(\{c\},\{d\})}), (p_{(\{c\},\{d\})}, d),$$

[1] Nevertheless, the α-algorithm may construct a Petri net that is not a WF-net (see, for instance, Fig. 6.12). Later, we will discuss such problems in detail.

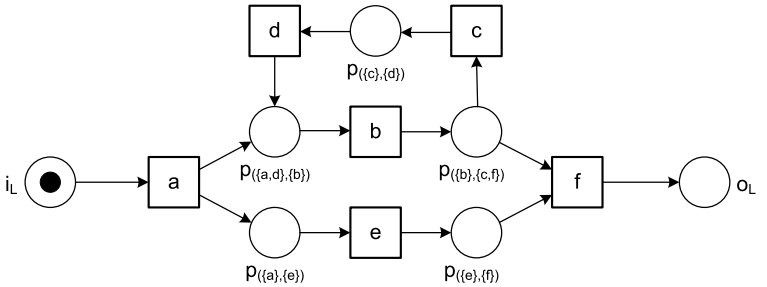


Fig. 6.8 WF-net N_5 derived from $L_5 = [\langle a, b, e, f\rangle^2, \langle a, b, e, c, d, b, f\rangle^3, \langle a, b, c, e, d, b, f\rangle^2, \langle a, b, c, d, e, b, f\rangle^4, \langle a, e, b, c, d, b, f\rangle^3]$

$$
\begin{aligned}
&(e, p_{(\{e\},\{f\})}), (p_{(\{e\},\{f\})}, f), (a, p_{(\{a,d\},\{b\})}), (d, p_{(\{a,d\},\{b\})}),\\
&(p_{(\{a,d\},\{b\})}, b), (b, p_{(\{b\},\{c,f\})}), (p_{(\{b\},\{c,f\})}, c), (p_{(\{b\},\{c,f\})}, f),\\
&(i_L, a), (f, o_L)\big\}\\
\alpha(L) = (P_L, T_L, F_L)&
\end{aligned}
$$

Figure 6.8 shows $N_5 = \alpha(L_5)$, i.e., the model just computed. N_5 can indeed replay the traces in L_5. Place names are not shown in Fig. 6.8, and we will also not show them in later WF-nets, because they can be derived from the surrounding transition names and just clutter the diagram.

6.2.3 Limitations of the α-Algorithm

In [157], it was shown that the α-algorithm can discover a large class of WF-nets if one assumes that the log is *complete* with respect to the log-based ordering relation $>_L$. This assumption implies that, for any complete event log L, $a >_L b$ if a can be directly followed by b. Consequently, a footprint like the one shown in Table 6.5 is assumed to be valid. We revisit the notion of completeness later in this chapter.

Even if we assume that the log is complete, the α-algorithm has some problems. There are many different WF-nets that have the same possible behavior, i.e., two models can be structurally different but trace equivalent. Consider, for instance, the following event log:

$$L_6 = \big[\langle a, c, e, g\rangle^2, \langle a, e, c, g\rangle^3, \langle b, d, f, g\rangle^2, \langle b, f, d, g\rangle^4\big]$$

$\alpha(L_6)$ is shown in Fig. 6.9. Although the model is able to generate the observed behavior, the resulting WF-net is needlessly complex. Two of the input places of g are redundant, i.e., they can be removed without changing the behavior. The places denoted as p_1 and p_2 are so-called *implicit places* and can be removed without

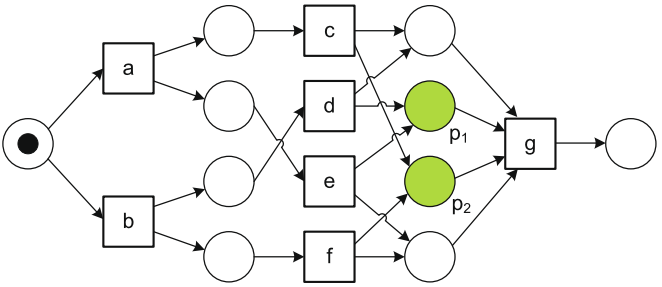


Fig. 6.9 WF-net N_6 derived from $L_6 = [\langle a, c, e, g\rangle^2, \langle a, e, c, g\rangle^3, \langle b, d, f, g\rangle^2, \langle b, f, d, g\rangle^4]$. The two highlighted places are redundant, i.e., removing them will simplify the model without changing its behavior

Fig. 6.10 Incorrect WF-net N_7 derived from $L_7 = [\langle a, c\rangle^2, \langle a, b, c\rangle^3, \langle a, b, b, c\rangle^2, \langle a, b, b, b, b, c\rangle^1]$

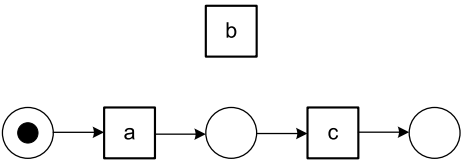
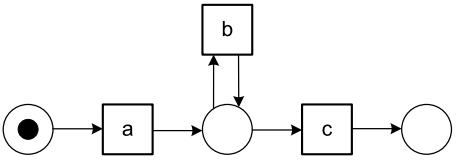


Fig. 6.11 WF-net N_7' having a so-called "short-loop" of length one



affecting the set of possible firing sequences. In fact, Fig. 6.9 shows only one of many possible trace equivalent WF-nets.

The original α-algorithm (as presented in Sect. 6.2.2) has problems dealing with short loops, i.e., loops of length one or two. For a loop of length one, this is illustrated by WF-net N_7 in Fig. 6.10, which shows the result of applying the basic algorithm to L_7,

$$L_7 = \left[\langle a, c\rangle^2, \langle a, b, c\rangle^3, \langle a, b, b, c\rangle^2, \langle a, b, b, b, b, c\rangle^1\right]$$

The resulting model is not a WF-net as transition b is disconnected from the rest of the model. The models allows for the execution of b before a and after c. This is not consistent with the event log. This problem can be addressed easily as shown in [11]. Using an improved version of the α-algorithm one can discover the WF-net shown in Fig. 6.11.

The problem with loops of length two is illustrated by Petri net N_8 in Fig. 6.12 which shows the result of applying the basic algorithm to L_8,

$$L_8 = \left[\langle a, b, d\rangle^3, \langle a, b, c, b, d\rangle^2, \langle a, b, c, b, c, b, d\rangle\right]$$

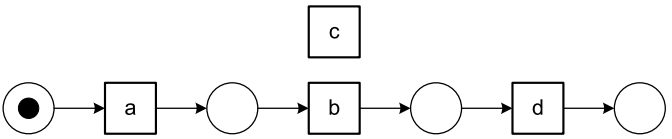


Fig. 6.12 Incorrect WF-net N_8 derived from $L_8 = [\langle a, b, d\rangle^3, \langle a, b, c, b, d\rangle^2, \langle a, b, c, b, c, b, d\rangle]$

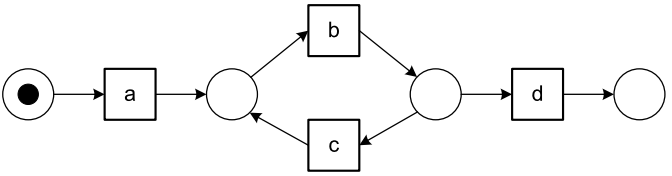


Fig. 6.13 Corrected WF-net N_8' having a so-called "short-loop" of length two

The following log-based ordering relations are derived from this event log: $a \rightarrow_{L_8} b$, $b \rightarrow_{L_8} d$, and $b \|_{L_8} c$. Hence the basic algorithm incorrectly assumes that b and c are in parallel because they follow one another. The model shown in Fig. 6.12 is not even a WF-net because c is not on a path from source to sink. Using the extension described in [11], the improved α-algorithm correctly discovers the WF-net shown in Fig. 6.13.

There are various ways to improve the basic α-algorithm to be able to deal with loops. The α^+-algorithm described in [11] is one of several alternatives to address problems related to the original algorithm presented in Sect. 6.2.2. The α^+-algorithm uses a pre- and post-processing phase. The pre-processing phase deals with loops of length two whereas the pre-processing phase inserts loops of length one.

The basic algorithm has no problems mining loops of length three or more. For a loop of involving at least three activities (say a, b, and c), concurrency can be distinguished from loops using relation $>_L$. For a loop we find only $a >_L b$, $b >_L c$, and $c >_L a$. If the three activities are concurrent, we find $a >_L b$, $a >_L c$, $b >_L a$, $b >_L c$, $c >_L a$, and $c >_L b$. Hence, it is easy to detect the difference. Note that for a loop of length two this is not the case. For a loop involving a and b, we find $a >_L b$ and $b >_L a$. If a and b are concurrent, we find the same relations. Hence, both constructs leave the same footprint in the event log.

A more difficult problem is the discovery of so-called *non-local dependencies* resulting from non-free choice process constructs. An example is shown in Fig. 6.14. This net would be a good candidate after observing the following event log:

$$L_9 = \left[\langle a, c, d\rangle^{45}, \langle b, c, e\rangle^{42}\right]$$

However, the α-algorithm will derive the WF-net without the places labeled p_1 and p_2. Hence, $\alpha(L_9) = N_4$, as shown in Fig. 6.6, although the traces $\langle a, c, e\rangle$ and $\langle b, c, d\rangle$ do not appear in L_9. Such problems can be (partially) resolved using refined versions of the α-algorithm such as the one presented in [185].

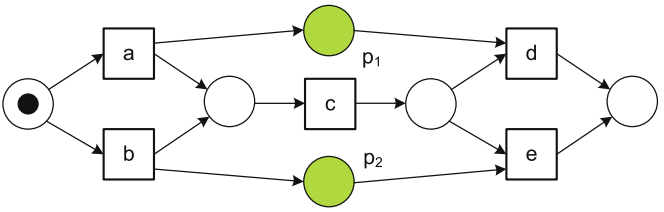


Fig. 6.14 WF-net N_9 having a non-local dependency

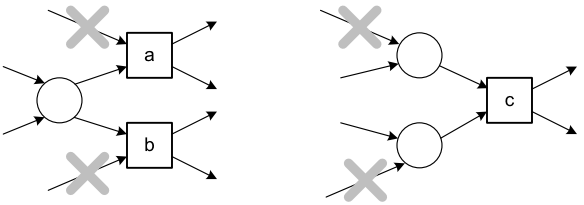


Fig. 6.15 Two constructs that may jeopardize the correctness of the discovered WF-net

Another limitation of the α-algorithm is that *frequencies are not taken into account*. Therefore, the algorithm is very sensitive to noise and incompleteness (see Sect. 6.4.2).

The α-algorithm is able to discover a large class of models. The basic 8-line algorithm has some limitations when it comes to particular process patterns (e.g., short-loops and non-local dependencies). Some of these problems can be solved using various refinements. As shown in [11, 157], the α-algorithm guarantees to produce a correct process model provided that the underlying process can be described by a WF-net that does not contain duplicate activities (two transitions with the same activity label) and silent transitions (activities that are not recorded in the event log), and does not use the two constructs shown in Fig. 6.15. See [11, 157] for the precise requirements.

Even if the underlying process is using constructs as shown in Fig. 6.15, the α-algorithm may still produce a useful process model. For instance, the α-algorithm is unable to discover the highlighted places (p_1 and p_2) in Fig. 6.14, but still produces a sound process model that is able to replay the log.

6.2.4 Taking the Transactional Life-Cycle into Account

When describing the typical information in event logs in Chap. 5, we discussed the *transactional life-cycle model* of an *activity instance*. Figure 5.3 shows examples of transaction types, e.g., schedule, start, complete, and suspend. Events often have such a transaction type attribute, e.g., $\#_{trans}(e) = complete$. The standard life-cycle extension of XES also provides such an attribute. The α-algorithm can be easily adapted to take this information into account. First of all, the log could be projected

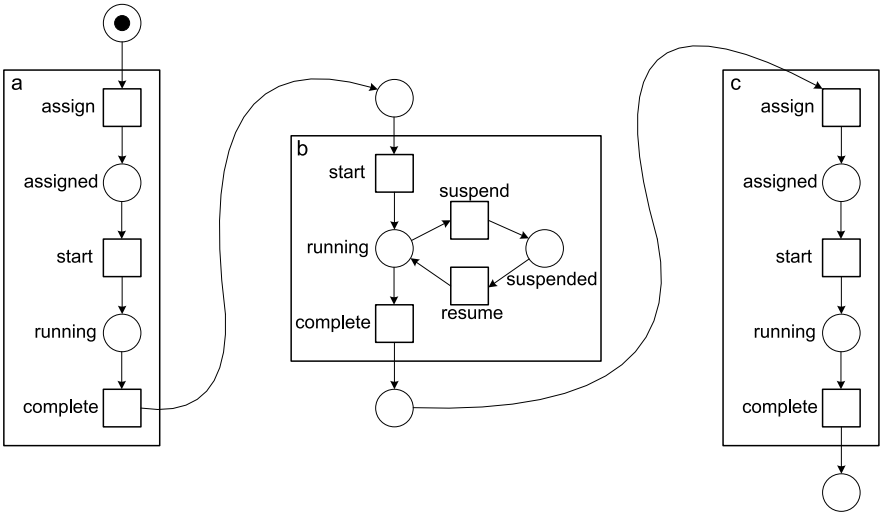


Fig. 6.16 Mining event logs with transactional information; the life-cycle of each activity is represented as a subprocess

onto smaller event logs in which each of the smaller logs contains all events related to a specific activity. This information can be used to discover the transactional life-cycle for each activity. Second, when mining the overall process, information about the general transactional life-cycle (e.g., Fig. 5.3) or information about an activity-specific transactional life-cycle can be exploited. Figure 6.16 illustrates the latter. All events related to an activity are mapped onto transitions embedded in a subprocess. The relations between the transitions for each subprocess are either discovered separately or modeled using domain knowledge. Figure 6.16 shows a sequence of three activities. Activities *a* and *c* share a common transactional life-cycle involving the event types *assign*, *start*, and *complete*. Activity *b* has a transactional life-cycle involving the event types *start*, *suspend*, *resume*, and *complete*.

6.3 Rediscovering Process Models

In Chap. 8, we will describe *conformance checking* techniques for measuring the quality of a process model *with respect to an event log*. However, when discussing the results of the α-algorithm, we already concluded that some WF-nets "could not be discovered" based on an event log. This assumes that we aim to discover a particular, known, model. In reality, we often do not know the "real" model. In fact, in practice, there is no such thing as *the* model describing a process. There may be many models (i.e., views on the same reality) and the process being studied may change while being discovered. However, as sketched in Fig. 6.17, we can create the experimental setting for testing process discovery algorithms in which we assume the original model to be known.

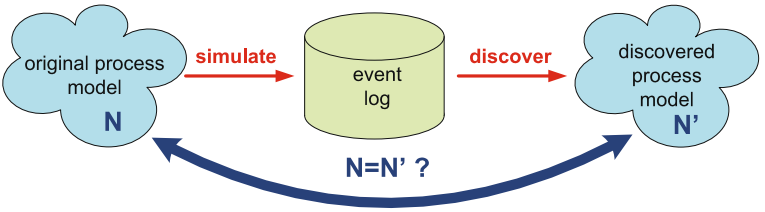


Fig. 6.17 The rediscovery problem: Is the discovered model N' equivalent to the original model N?

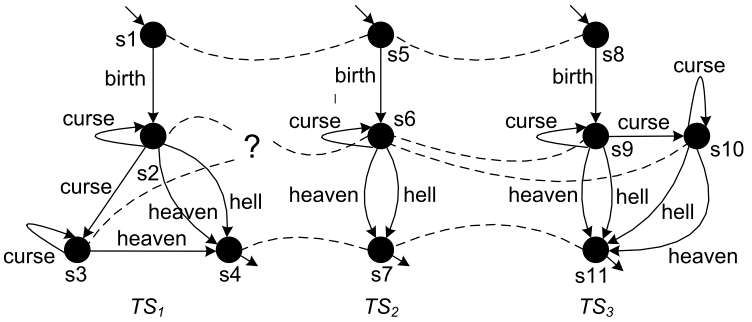


Fig. 6.18 Three trace equivalent transition systems: TS_1 and TS_2 are not bisimilar, but TS_2 and TS_3 are bisimilar

Starting point in Fig. 6.17 is a process model, e.g., a WF-net N. Based on this model we can run many simulation experiments and record the simulated events in an event log. Let us assume that the event log is complete with respect to some criterion, e.g., if x can be followed by y in N it happened at least once according to log. Using the complete event log as input for a process discovery algorithm (e.g., the α-algorithm), we can construct a new model. Now the question is: "What do the discovered model N' and the original model N have in common? Are they equivalent?" Equivalence can be viewed at different levels. For example, it is unreasonable to expect that a discovery algorithm is able to reconstruct the original layout as this information is not in the log; layout information is irrelevant for the behavior of a process. For the same reason, it is unreasonable to expect that the original place names of the WF-net can be reconstructed. The α-algorithm generates places named $p_{(A,B)}$. These are of course not intended to match original place names. Therefore, we need to focus on *behavior* (and not on layout and syntax) when comparing the discovered model N' and the original model N.

Three notions of behavioral equivalence
As shown in [176], many equivalence notions can be defined. Here, we informally describe three well-known notions: *trace equivalence*, *bisimilarity*,

and *branching bisimilarity*. These notions are defined for a pair of transition systems TS_1 and TS_2 (Sect. 3.2.1) and not for higher-level languages such as WF-nets, BPMN, EPCs, and YAWL. However, any model with executable semantics can be transformed into a transition system. Therefore, we can assume that the original process model N and the discovered process model N' mentioned in Fig. 6.17 define two transition systems that can be used as a basis for comparison.

Trace equivalence considers two transition systems to be equivalent if their sets of execution sequences are identical. Let TS_2 be the transition system corresponding to WF-net $N_6 = \alpha(L_6)$ shown in Fig. 6.9 and let TS_1 be the transition system corresponding to the same WF-net but now without places p_1 and p_2. Although both WF-nets are syntactically different, the sets of execution sequences of TS_1 and TS_2 are the same. However, two transition systems that allow for the same set of execution sequences may also be quite different as illustrated by Fig. 6.18.

The three transition systems in Fig. 6.18 are trace equivalent: any trace in one transition system is also possible in any of the other transition systems. For instance, the trace $\langle birth, curse, curse, curse, heaven \rangle$ is possible in all three transition systems. However, there is a relevant difference between TS_1 and TS_2. In TS_1 one can end up in state $s3$ where one will always go to heaven despite the cursing. Such a state does not exist in TS_2; while cursing in state $s6$ one can still go to hell. When moving from state $s2$ to state $s3$ in TS_1 a choice was made which cannot be seen in the set of traces but that is highly relevant for understanding the process.

Bisimulation equivalence, or bisimilarity for short, is a more refined notion taking into account the moment of choice. Two transition systems are bisimilar if the first system can "mimic any move" of the second, and vice versa (using the same relation). Consider, for example, TS_2 and TS_3 in Fig. 6.18. TS_2 can simulate TS_3 and vice versa. The states of both transition systems are related by dashed lines; $s5$ is related to $s8$, $s6$ is related to both $s9$ and $s10$, and $s7$ is related to $s11$. In two related states the same set of actions needs to be possible and taking any of these actions on one side should lead to a related state when taking the same action on the other side. Because TS_2 can move from $s5$ to $s6$ via action *birth*, TS_3 should also be able to take a *birth* action in $s8$ resulting in a related state ($s9$). TS_2 and TS_3 are bisimilar because any action by one can be mimicked by the other. Now consider TS_1 and TS_2. Here, it is impossible to relate $s3$ in TS_1 to a corresponding state in TS_2. If $s3$ is related to $s6$, then in $s3$ it should be possible to do a *hell* action, but this is not the case. Hence, TS_2 can simulate TS_1, i.e., any action in TS_1 can be mimicked by TS_2, but TS_1 *cannot* simulate TS_2. Therefore, TS_1 and TS_2 are not bisimilar. Bisimulation equivalence is a stronger equivalence relation than trace equivalence, i.e., if two transition systems are bisimilar, then they are also trace equivalent.

Branching bisimulation equivalence, or branching bisimilarity for short, takes *silent actions* into account. In Chap. 3 we introduced already the label τ for this purpose. A τ action is "invisible", i.e., cannot be observed. In terms of process mining this means that the corresponding activity is not recorded in the event log. As before, two transition systems are branching bisimilar if the first system can "follow any move" of the second and vice versa, but now taking τ actions into account. (Here, we do not address subtle differences between weak bisimulation, also known as observational equivalence, and branching bisimulation equivalence [176].) If one system takes a τ action, then the second system may also take a τ action or do nothing (as long as the states between both systems remain related). If one system takes a non-τ action, then the second system should also be able to take the same non-τ action possibly preceded by sequence of τ actions. The states before and after the non-τ action, need to be related. Figure 6.19 shows two YAWL models and their corresponding transition systems TS_1 and TS_2. The two transition systems are *not* branching bisimilar. The reason is that in the YAWL model on the left, a choice is made after task *check*, whereas in the other model the choice is postponed until either *reject* or *accept* happens. Therefore, the YAWL model on the left cannot simulate the model on the right. Technically, states $s3$ and $s4$ in TS_1 do not have a corresponding state in TS_2. It is impossible to relate $s3$ and $s4$ to $s7$ since $s7$ allows for both actions whereas $s3$ and $s4$ allow for only one action. The YAWL model on the right models the so-called *deferred choice* workflow pattern whereas the YAWL model on the left models the more common *exclusive choice* pattern [155].

Branching bisimulation equivalence is highly relevant for process mining since typically not all actions are recorded in the event log. For example, if the choice made in task *check* is not recorded in the event log, then one discovers the YAWL model on the right, i.e., the right moment of choice cannot be captured.

Although both models in Fig. 6.19 are not branching bisimilar they are trace equivalent. In both models there are only two possible (visible) traces: $\langle check, reject \rangle$ and $\langle check, accept \rangle$.

We refer to [176] for formal definitions of the preceding concepts. Here we discuss these concepts because they are quite important when judging process mining results.

The different notions of equivalence show that the comparison of the original model and the discovered model in Fig. 6.17 is not a simple syntactical check. Instead a choice must be made with respect to the type of behavioral equivalence that is appropriate.

As mentioned before, the experimental setting shown in Fig. 6.17 can only be used in the situation in which the model is known beforehand. In most applica-

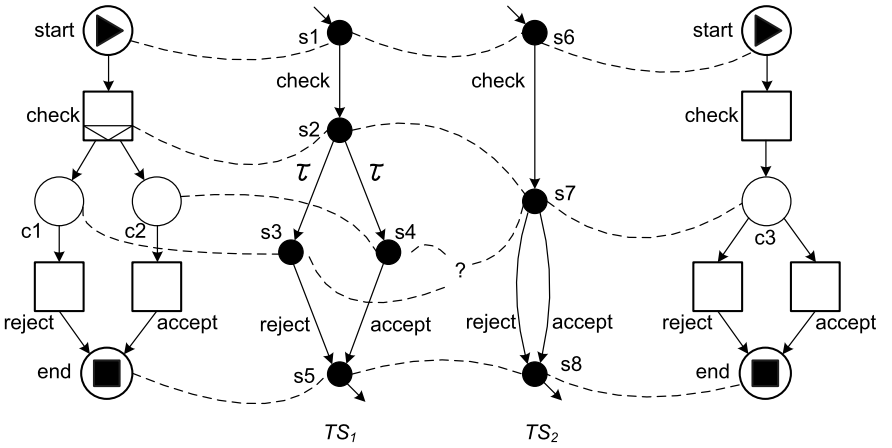


Fig. 6.19 Two YAWL models and the corresponding transition systems

tions such a model is not known. Moreover, classical notions such as trace equivalence, bisimilarity, and branching bisimilarity provide only true/false answers. As discussed in [14], a binary equivalence is not very useful in the context of process mining. If two processes are very similar (identical except for some exceptional paths), classical equivalence checks will simply conclude that the processes are not equivalent rather than stating that the processes are, e.g., 95% similar. Therefore, this book will focus on the comparison of a model and an event log rather than comparing two models. For instance, in Chap. 8 we will show techniques that can conclude that 95% of the event log "fits" the model.

6.4 Challenges

The α-algorithm was one of the first process discovery algorithms to adequately capture concurrency (see also Sect. 7.6). Today there are much better algorithms that overcome the weaknesses of the α-algorithm. These are either variants of the α-algorithm or algorithms that use a completely different approach, e.g., genetic mining or synthesis based on regions. In Chap. 7, we review some of these alternative approaches. However, before presenting new process discovery techniques, we first elaborate on the main challenges. For this purpose we show the effect that a representational bias can have (Sect. 6.4.1). Then we discuss problems related to the input event log that may be noisy or incomplete (Sect. 6.4.2). In Sect. 6.4.3, we discuss the four quality criteria mentioned earlier: fitness, precision, generalization, and simplicity. Finally, Sect. 6.4.4 again emphasizes that discovered models are just a view on reality. Hence, the usefulness of the model strongly depends on the questions one seeks to answer.

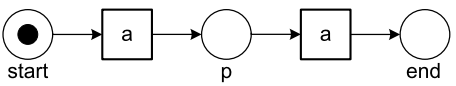


Fig. 6.20 A WF-net having two transitions with the same label describing event log $L_{10} = [\langle a, a \rangle^{55}]$

6.4.1 Representational Bias

At the beginning of the chapter we decided to focus on a mining algorithm that produces a WF-net, i.e., we assumed that the underlying process can be adequately described by a WF-net. Any discovery technique requires such a *representational bias*. For example, algorithms for learning decision trees (see Sect. 4.2) make similar assumptions about the structure of the resulting tree. For instance, most decision tree learners can only split once on an attribute on every path in the tree.

When discussing the α-algorithm we assumed that the process to be discovered is a sound WF-net. More specifically, we assumed that the underlying process can be described by a WF-net where each transition bears a unique and visible label. In such a WF-net it is not possible to have two transitions with the same label (i.e., $l(t_1) = l(t_2)$ implies $t_1 = t_2$) or transitions whose occurrences remain invisible (i.e., it is not possible to have a so-called silent transition, so for all transitions $t, l(t) \neq \tau$). (See Sect. 3.2.2 and the earlier discussion on branching bisimulation equivalence.) These assumptions may seem harmless, but have a noticeable effect on the class of process models that can be discovered. We show two examples illustrating the impact of such a representational bias.

For an event log like $L_{10} = [\langle a, a \rangle^{55}]$, i.e., for all cases precisely two a's are executed, ideally one would like to discover the WF-net shown in Fig. 6.20. Unfortunately, this process model will not be discovered due to the representational bias of the α-algorithm. There is no WF-net without duplicate and τ labels that has the desired behavior and the α-algorithm can only discover such WF-nets (i.e., each transition needs to have unique visible label).

Let us now consider event log $L_{11} = [\langle a, b, c \rangle^{20}, \langle a, c \rangle^{30}]$. Figure 6.21(a) describes the underlying process well: activity b can be skipped by executing the τ transition. Figure 6.21(b) shows an alternative WF-net using two a transitions and no τ transition. These two models are trace equivalent. (They are not branching bisimilar because the moment of choice is different.) However, it is not possible to construct a WF-net without duplicate and τ labels that is trace equivalent to these two models. Figure 6.21(c) shows the model produced by the α-algorithm; because of the representational bias, the algorithm is destined to fail for this log. The WF-net in Fig. 6.21(c) can only reproduce trace $\langle a, b, c \rangle$ and not $\langle a, c \rangle$.

Event logs L_{10} and L_{11} illustrate the effect a representational bias can have. However, from the viewpoint of the α-algorithm, the choice to not consider duplicate labels and τ transitions is sensible. τ transitions are not recorded in the log and hence any algorithm will have problems reconstructing their behavior. Multiple transitions with the same label are undistinguishable in the event log. Therefore, any algorithm will have problems associating the corresponding events to one of these transitions.

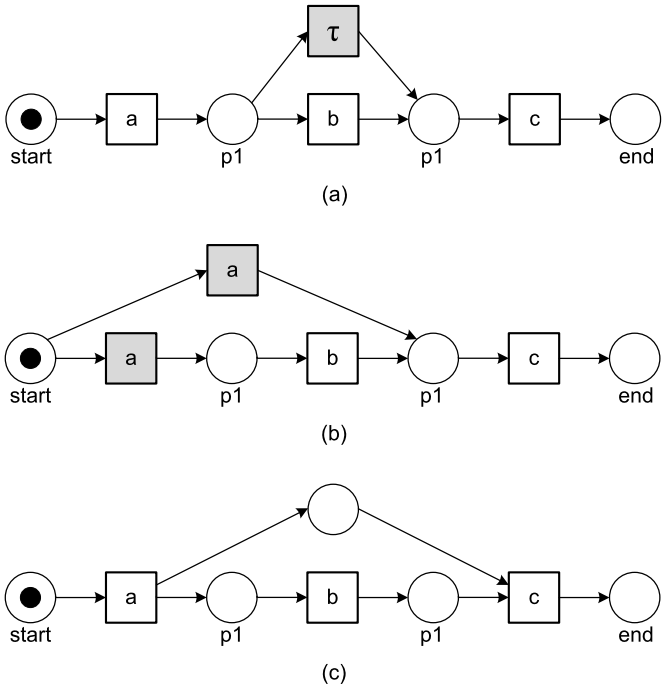


Fig. 6.21 Three WF-nets for the event log $L_{11} = [\langle a, b, c\rangle^{20}, \langle a, c\rangle^{30}]$

The problems sketched previously apply to many process discovery algorithms. For example, the choice between the concurrent execution of b and c or the execution of just e shown in Fig. 6.1 cannot be handled by many algorithms. Most algorithms do *not* allow for so-called "non-free-choice constructs" where concurrency and choice meet. The concept of *free-choice nets* is well-defined in the Petri net domain [45]. A Petri net is free choice if any two transitions sharing an input place have identical input sets, i.e., $\bullet t_1 \cap \bullet t_2 \neq \emptyset$ implies $\bullet t_1 = \bullet t_2$ for any $t_1, t_2 \in T$. Most analysis questions (e.g., soundness) can be answered in polynomial time for free-choice nets [136, 168]. Moreover, many process modeling languages are inherently free-choice, thus making this an interesting subclass. Unfortunately, in reality processes tend to be non-free-choice. The example of Fig. 6.1 shows that sometimes the α-algorithm is able to deal with non-free-choice constructs. However, there are many non-free-choice processes that cannot be discovered by the α-algorithm (see for example N_9 in Fig. 6.14). The non-free-choice construct is just one of many constructs that existing process mining algorithms have problems with. Other examples are arbitrary nested loops, cancelation, unbalanced splits and joins, and partial synchronization. In this context it is important to observe *process discovery is, by definition, restricted by the expressive power of the target language*, i.e., the representational bias.

For the reader interested in the topic, we refer to the *workflow patterns* [155, 191] mentioned earlier. These patterns help to discuss and identify the representational bias of a language.

The representational bias helps limiting the search space of possible candidate models. This can make discovery algorithms more efficient. However, it can also be used to give preference to particular types of models. It seems that existing approaches can benefit from selecting a more suitable representational bias. For instance, the α-algorithm may yield models that have deadlocks or livelocks. Here it would be nice to have a representational bias to limit the search space to only sound models (i.e., free of deadlocks and other anomalies). Unfortunately, currently, this can typically only be achieved by severely limiting the expressiveness of the modeling language or by using more time-consuming analysis techniques. Consider, for example, the so-called *block-structured* process models. A model is block-structured if it satisfies a number of syntactical requirements such that soundness is guaranteed by these requirements. Different definitions exist [49, 132, 187]. Most of these definitions require a one-to-one correspondence between splits and joins, e.g., concurrent paths created by an AND-split need to be synchronized by the corresponding AND-join. Since many real-life processes are not block structured (see for example Figs. 14.1 and 14.10), one should be careful to not limit the expressiveness too much. Note that techniques that turn unstructured models into block-structured process models tend to introduce many duplicate or silent activities. Therefore, such transformations do not alleviate the core problems.

6.4.2 Noise and Incompleteness

To discover a suitable process model it is assumed that the event log contains a *representative sample of behavior*. Besides the issues mentioned in Chap. 5 (e.g., correlating events and scoping the log), there are two related phenomena that may make an event log less representative for the process being studied:

- (*Noise*) The event log contains rare and infrequent behavior not representative for the typical behavior of the process.[2]
- (*Incompleteness*) The event log contains too few events to be able to discover some of the underlying control-flow structures.

6.4.2.1 Noise

Noise, as defined in this book, does not refer to incorrect logging. When extracting event logs from various data sources one needs to try to locate data problems

[2]Note that the definition of noise may be a bit counter-intuitive. Sometimes the term "noise" is used to refer to incorrectly logged events, i.e., errors that occurred while recording the events. Such a definition is not very meaningful as no event log will explicitly reveal such errors. Hence, we consider "outliers" as noise. Moreover, we assume that such outliers correspond to exceptional behavior rather than logging errors.

as early as possible. However, at some stage one needs to assume that the event log contains information on what really happened. It is impossible for a discovery algorithm do distinguish incorrect logging from exceptional events. This requires human judgment and pre- and postprocessing of the log. Therefore, we use the term "noise" to refer to rare and infrequent behavior ("outliers") rather than errors related to event logging. For process mining it is important to filter out noise and several process discovery approaches specialize in doing so, e.g., heuristic mining, genetic mining, and fuzzy mining.

Recall the *support* and *confidence* metrics defined in the context of learning association rules. The support of a rule $X \Rightarrow Y$ indicates the applicability of the rule, i.e., the fraction of instances for which with both antecedent and consequent hold. The confidence of a rule $X \Rightarrow Y$ indicates the reliability of the rule. If rule $tea \wedge latte \Rightarrow muffin$ has a support of 0.2 and a confidence of 0.9, then 20% of the customers actually order tea, latte and muffins at the same time and 90% of the customers that order tea and latte also order a muffin. For learning association rules we defined a threshold for both confidence and support, i.e., rules with low confidence or support are considered to be noise.

Let us informally apply the idea of confidence and support to the basic α-algorithm. Starting point for the α-algorithm is the $>_L$ relation. Recall that $a >_L b$ if and only if there is a trace in L in which a is directly followed by b. Now we can define the support of $a >_L b$ based on number of times the pattern $\langle \ldots, a, b, \ldots \rangle$ appears in the log, e.g., the fraction of cases in which the pattern occurs. Subsequently, we can use a threshold for cleaning the $>_L$ relation. The confidence of $a >_L b$ can be defined by comparing the number of times the pattern $\langle \ldots, a, b, \ldots \rangle$ appears in the log divided by the frequency of a and b. For example, suppose that $a >_L b$ has a reasonable support, e.g., the pattern $\langle \ldots, a, b, \ldots \rangle$ occurs 1000 times in the log. Moreover, a occurs 1500 times and b occurs 1200 times. Clearly, $a >_L b$ has a good confidence. However, if the pattern $\langle \ldots, a, b, \ldots \rangle$ occurs 1000 times and a and b are very frequent and occur each more than 100,000 times , then the confidence in $a >_L b$ is much lower. The $>_L$ relation is the basis for the footprint matrices as shown in Tables 6.1, 6.2, 6.3, and 6.5. Hence, by removing "noisy $a >_L b$ rules", we obtain a more representative footprint, and a better starting point for the α-algorithm. (There are several complications when doing this, however, the basic idea should be clear.) This simplified discussion shows how "noise" can be quantified and addressed when discovering process models. When presenting heuristic mining in Sect. 7.2 we return to this topic.

In the context of noise, we also talk about the *80/20 model*. Often we are interested in the process model that can describe 80% of the behavior seen in the log. This model is typically relatively simple because the remaining 20% of the log account for 80% of the variability in the process.

6.4.2.2 Incompleteness

When it comes to process mining the notion of *completeness* is also very important. It is related to noise. However, whereas noise refers to the problem of having "too

much data" (describing rare behavior), completeness refers to the problem of having "too little data".

Like in any data mining or machine learning context one cannot assume to have seen all possibilities in the "training material" (i.e., the event log at hand). For WF-net N_1 in Fig. 6.1 and event log $L_1 = [\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^2, \langle a, e, d\rangle]$, the set of possible traces found in the log is exactly the same as the set of possible traces in the model. In general, this is not the case. For instance, the trace $\langle a, b, e, c, d\rangle$ may be possible but did not (yet) occur in the log. Process models typically allow for an exponential or even infinite number of different traces (in case of loops). Moreover, some traces may have a much lower probability than others. Therefore, it is unrealistic to assume that every possible trace is present in the event log.

The α-algorithm assumes a relatively weak notion of completeness to avoid this problem. Although N_3 has infinitely many possible firing sequences, a small log like $L_3 = [\langle a, b, c, d, e, f, b, d, c, e, g\rangle, \langle a, b, d, c, e, g\rangle^2, \langle a, b, c, d, e, f, b, c, d, e, f, b, d, c, e, g\rangle]$ can be used to construct N_3. The α-algorithm uses a local completeness notion based on $>_L$, i.e., if there are two activities a and b, and a can be directly followed by b, then this should be observed at least once in the log.

To illustrate the relevance of completeness, consider a process consisting of 10 activities that can be executed in parallel and a corresponding log that contains information about 10,000 cases. The total number of possible interleavings in the model with 10 concurrent activities is $10! = 3{,}628{,}800$. Hence, it is impossible that each interleaving is present in the log as there are fewer cases (10,000) than potential traces (3,628,800). Even if there are 3,628,800 cases in the log, it is extremely unlikely that all possible variations are present. To motivate this consider the following analogy. In a group of 365 people it is very unlikely that everyone has a different birthdate. The probability is $365!/365^{365} \approx 1.454955 \times 10^{-157} \approx 0$, i.e., incredibly small. The number of atoms in the universe is often estimated to be approximately 10^{79} [189]. Hence, the probability of picking a particular atom from the entire universe is much higher than covering all 365 days. Similarly, it is unlikely that all possible traces will occur for any process of some complexity because most processes have much more than 365 possible execution paths. In fact, because typically some sequences are less probable than others, the probability of finding all traces is even smaller. Therefore, weaker completeness notions are needed. For the process in which 10 activities can be executed in parallel, local completeness can reduce the required number of observations dramatically. For example, for the α-algorithm only $10 \times (10 - 1) = 90$ rather than 3,628,800 different observations are needed to construct the model.

6.4.2.3 Cross-Validation

The preceding discussion on completeness and noise shows the need for *cross-validation* as discussed in Sect. 4.6.2. The event log can be split into a *training log* and a *test log*. The training log is used to learn a process model whereas the test log is used to evaluate this model based on unseen cases. Chapter 8 will present con-

crete techniques for evaluating the quality of a model with respect to an event log. For example, if many traces of the test log do not correspond to possible firing sequences of the WF-net discovered based on the training log, then one can conclude that the quality of the model is low.

Also *k-fold cross-validation* can be used, i.e., the event log is split into k equal parts, e.g., $k = 10$. Then k tests are done. In each test, one of the subsets serves as a test log whereas the other $k - 1$ subsets serve together as the training log.

One of the problems for cross validation is the lack of negative examples, i.e., the log only provides examples of possible behavior and does not provide explicit examples describing scenarios that are impossible (see discussion in Sect. 4.6.3). This is complicating cross-validation. One possibility is to insert artificially generated negative events [59, 60, 122]. The basic idea is to compare the quality of the discovered model with respect to the test log containing *actual behavior* with the quality of the discovered model with respect to a test log containing *random behavior*. Ideally, the model scores much better on the log containing actual behavior than on the log containing random behavior.

Cross-validation can also be applied at the level of the footprint matrix. Simply split the event log in k parts and construct the footprint matrix for each of the k parts. If the k footprint matrices are very different (even for smaller values of k), then one can be sure that the event log does not meet the completeness requirement imposed by the α-algorithm. Such a validation can be done before constructing the process model. If there are strong indications that $>_L$ is far from complete, more advanced process mining techniques need to be applied and the results need to be interpreted with care (see also Chap. 7).

6.4.3 Four Competing Quality Criteria

Completeness and noise refer to qualities of the event log and do not say much about the quality of the discovered model. Determining the quality of a process mining result is difficult and is characterized by many dimensions. In this book, we refer to four main quality dimensions: *fitness*, *simplicity*, *precision*, and *generalization*. In this section, we review these four dimensions without providing concrete metrics. Some of the dimensions will be discussed in later chapters in more detail. However, after reading this section it should already be clear that they can indeed be quantified.

Figure 6.22 gives a high-level characterization of the four quality dimensions. A model with good *fitness* allows for the behavior seen in the event log. A model has a perfect fitness if all traces in the log can be replayed by the model from beginning to end. There are various ways of defining fitness. It can be defined at the case level, e.g., the fraction of traces in the log that can be fully replayed. It can also be defined at the event level, e.g., the fraction of events in the log that are indeed possible according to the model. When defining fitness many design decisions need to be made. For example: What is the penalty if a step needs to be skipped and what is the penalty if tokens remain in the WF-net after replay? Later, we will give concrete definitions for fitness.

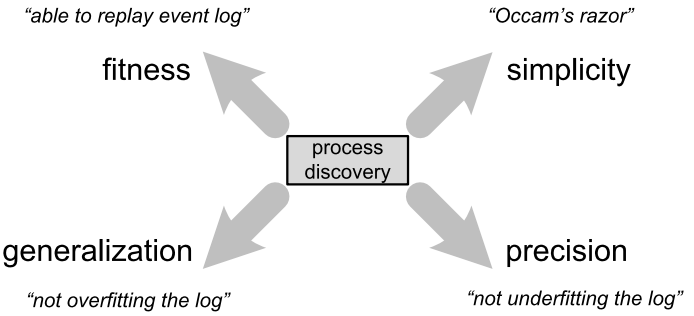


Fig. 6.22 Balancing the four quality dimensions: *fitness*, *simplicity*, *precision*, and *generalization*

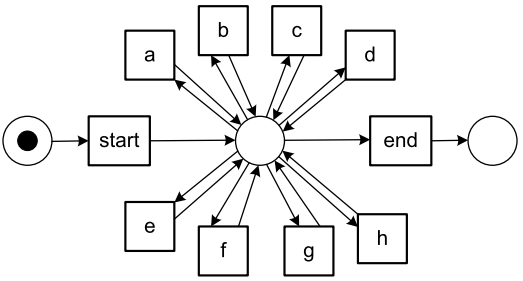


Fig. 6.23 The so-called "flower Petri net" allowing for any log containing activities $\{a, b, \ldots, h\}$

In Sect. 4.6.1, we defined performance measures like error, accuracy, tp-rate, fp-rate, precision, recall, and F1 score. Recall, also known as the tp-rate, measures the proportion of positive instances indeed classified as positive (tp/p). The traces in the log are positive instances. When such an instance can be replayed by the model, then the instance is indeed classified as positive. Hence, the various notions of fitness can be seen as variants of the recall measure. Most of the notions defined in Sect. 4.6.1 cannot be used because there are *no negative examples*, i.e., *fp* and *tn* are unknown (see Fig. 4.14). Since the event log does not contain information about events that could *not* happen at a particular point in time, other notations are needed.

The *simplicity* dimension refers to *Occam's Razor*. This principle was already discussed in Sect. 4.6.3. In the context of process discovery this means that the simplest model that can explain the behavior seen in the log, is the best model. The complexity of the model could be defined by the number of nodes and arcs in the underlying graph. Also more sophisticated metrics can be used, e.g., metrics that take the "structuredness" or "entropy" of the model into account. See [101] for an empirical evaluation of the *model complexity metrics* defined in literature. In Sect. 4.6.3, we also mentioned that this principle can be operationalized using the *Minimal Description Length* (MDL) principle [63, 190].

Fitness and simplicity alone are not adequate. This is illustrated by the so-called "flower model" shown in Fig. 6.23. The "flower Petri net" allows for any sequence starting with *start* and ending with *end* and containing any ordering of activities in between. Clearly, this model allows for all event logs used to introduce the

α-algorithm. The added *start* and *end* activities in Fig. 6.23 are just a technicality to turn the "flower model" into a WF-net. Surprisingly, all event logs shown thus far $(L_1, L_2, \ldots, L_{11})$ can be replayed by this single model. This shows that the model is not very useful. In fact, the "flower model" does not contain any knowledge other than the activities in the event log. The "flower model" can be constructed based on the occurrences of activities only. The resulting model is simple and has a perfect fitness. Based on the first two quality dimensions this model is acceptable. This shows that the fitness and simplicity criteria are necessary, but not sufficient.

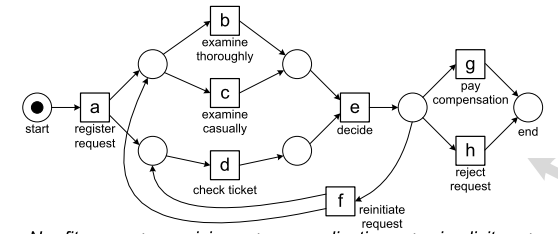
If the "flower model" is on one end of the spectrum, then the "enumerating model" is on the other end of the spectrum. The enumerating model of a log simply lists all the sequences possible, i.e., there is a separate sequential process fragment for each trace in the model. At the start there is one big XOR split selecting one of the sequences and at the end these sequences are joined using one big XOR join. If such a model is represented by a Petri net and all traces are unique, then the number of transitions is equal to the number of events in the log. The "enumerating model" is simply an encoding of the log. Such a model is complex but, like the "flower model", has a perfect fitness.

Extreme models such as the "flower model" (anything is possible) and the "enumerating model" (only the log is possible) show the need for two additional dimensions. A model is *precise* if it does not allow for "too much" behavior. Clearly, the "flower model" lacks precision. A model that is not precise is "underfitting". Underfitting is the problem that the model over-generalizes the example behavior in the log, i.e., the model allows for behaviors very different from what was seen in the log.

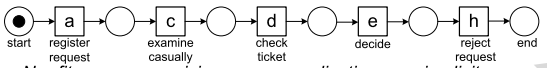
A model should *generalize* and not restrict behavior to the examples seen in the log (like the "enumerating model"). A model that does not generalize is "overfitting". Overfitting is the problem that a very specific model is generated whereas it is obvious that the log only holds example behavior, i.e., the model explains the particular sample log, but a next sample log of the same process may produce a completely different process model.

Process mining algorithms need to strike a balance between "overfitting" and "underfitting". A model is overfitting if it does not generalize and only allows for the exact behavior recorded in the log. This means that the corresponding mining technique assumes a very strong notion of completeness: "If the sequence is not in the event log, it is not possible!". An underfitting model over-generalizes the things seen in the log, i.e., it allows for more behavior even when there are no indications in the log that suggest this additional behavior (like in Fig. 6.23).

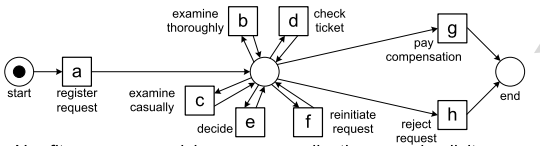
Let us now consider some examples showing that it is difficult to balance between being too general and too specific. Consider, for example, WF-net N_4 shown in Fig. 6.6 and N_9 shown in Fig. 6.14. Both nets can produce the log $L_9 = [\langle a, c, d\rangle^{45}, \langle b, c, e\rangle^{42}]$, but only N_4 can produce $L_4 = [\langle a, c, d\rangle^{45}, \langle b, c, d\rangle^{42}, \langle a, c, e\rangle^{38}, \langle b, c, e\rangle^{22}]$. Clearly, N_4 is the logical choice for L_4. Moreover, although both nets can produce L_9, it is obvious that N_9 is a better model for L_9 as none of the 87 cases follows one of the two additional paths ($\langle b, c, d\rangle$ and $\langle a, c, e\rangle$). However, now consider $L_{12} = [\langle a, c, d\rangle^{99}, \langle b, c, d\rangle^{1}, \langle a, c, e\rangle^{2}, \langle b, c, e\rangle^{98}]$. One can argue that



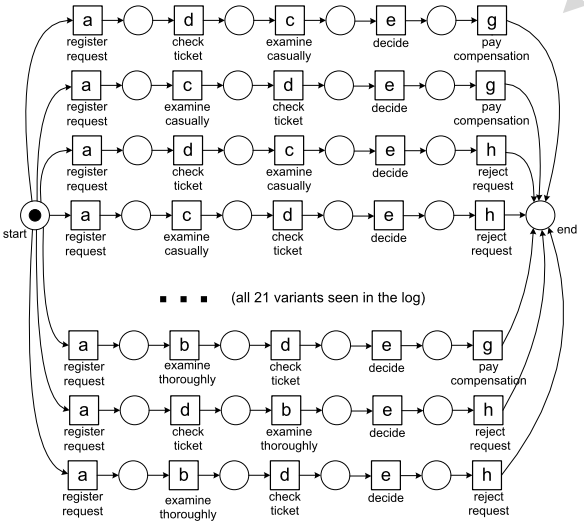
N_1 : *fitness* = +, *precision* = +, *generalization* = +, *simplicity* = +



N_2 : *fitness* = -, *precision* = +, *generalization* = -, *simplicity* = +



N_3 : *fitness* = +, *precision* = -, *generalization* = +, *simplicity* = +



N_4 : *fitness* = +, *precision* = +, *generalization* = -, *simplicity* = -

#	trace
455	acdeh
191	abdeg
177	adceh
144	abdeh
111	acdeg
82	adceg
56	adbeh
47	acdefdbeh
38	adbeg
33	acdefbdeh
14	acdefbdeg
11	acdefdbeg
9	adcefcdeh
8	adcefdbeh
5	adcefbdeg
3	acdefbdefdbeg
2	adcefdbeg
2	adcefbdefbdeg
1	adcefdbefbdeh
1	adbefbdefdbeg
1	adcefdbefcdefdbeg
1391	

Fig. 6.24 Four alternative models for the same log

N_4 is a better model for L_{12} as all traces can be reproduced. However, 197 out of 200 traces can be explained by the more precise model N_9. If the three traces are seen as noise, the main behavior is captured by N_9 and not N_4. Such considerations show that there is a delicate balance between "overfitting" and "underfitting". Hence, it is difficult, if not impossible, to select "the best" model.

Figure 6.24 illustrates the preceding discussion using the example from Chap. 2. Assume that the four models that are shown are discovered based on the event log also depicted in the figure. There are 1391 cases. Of these 1391 cases, 455 followed the trace $\langle a, c, d, e, h\rangle$. The second most frequent trace is $\langle a, b, d, e, g\rangle$ which was followed by 191 cases.

If we apply the α-algorithm to this event log, we obtain model N_1 shown in Fig. 6.24. A comparison of the WF-net N_1 and the log shows that this model is quite good; it is simple and has a good fitness. Moreover, it balances between overfitting and underfitting.

The other three models in Fig. 6.24 have problems with respect to one or more quality dimensions. WF-net N_2 models only the most frequent trace, i.e., it only allows for the sequence $\langle a, c, d, e, h\rangle$. Hence, none of the other $1391 - 455 = 936$ traces fits. Moreover, the model does not generalize, i.e., N_2 is also overfitting.

WF-net N_3 is a variant of the "flower model". Only the start and end transitions are captured well. The fitness is good, the model is simple, and not overfitting. However, N_3 lacks precision, i.e., is underfitting, as for example the trace $\langle a, b, b, b, b, b, b, f, f, f, f, f, g\rangle$ is possible. This behavior seems to be very different from any of the traces in the log.

Figure 6.24 shows only a part of WF-net N_4. This model simply enumerates the 21 different traces seen in the event log. This model is precise and has a good fitness. However, WF-net N_4 is overly complex and is overfitting.

The four models in Fig. 6.24 illustrate the four quality dimensions. Each of these dimensions can be quantified as shown in [121]. In [121], a replay technique is described to quantify fitness resulting in a value between 0 (very poor fitness) to 1 (perfect fitness). A notion called "structural appropriateness" considers the simplicity dimension; the model is analyzed to see whether it is "minimal in structure". Another notion called "behavioral appropriateness" analyzes the balance between overfitting and underfitting. There are different ways to operationalize the four quality dimensions shown in Fig. 6.22. Depending on the representational bias and goals of the analyst, different metrics can be quantified.

6.4.4 Taking the Right 2-D Slice of a 3-D Reality

The simple examples shown in this chapter already illustrate that process discovery is a non-trivial problem that requires sophisticated analysis techniques. Why is process mining such a difficult problem? There are obvious reasons that also apply to many other data mining and machine learning problems, e.g., dealing with noise and a complex and large search space. However, there are also some specific problems:

- There are *no negative examples* (i.e., a log shows what has happened but does not show what could not happen);
- Due to concurrency, loops, and choices the *search space has a complex structure* and the log typically contains only a *fraction* of all possible behaviors; and

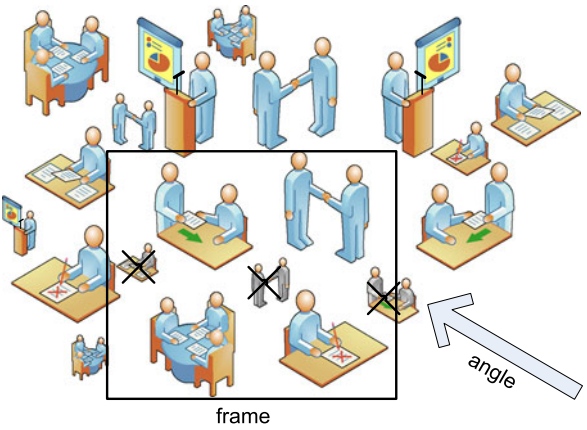


Fig. 6.25 Creating a 2-D slice of a 3-D reality: the process is *viewed* from a specific angle, the process is scoped using a *frame*, and the *resolution* determines the granularity of the resulting model

- There is *no clear relation between the size of a model and its behavior* (i.e., a smaller model may generate more or less behavior although classical analysis and evaluation methods typically assume some monotonicity property).

The next chapter will show several process discovery techniques that adequately address these problems.

As we will see in Part IV, the discovered process model is *just the starting point* for analysis. By relating events in the log to the discovered model, all kinds of analysis are possible, e.g., checking conformance, finding bottlenecks, optimizing resource allocation, reducing undesired variability, time prediction, and generating recommendations.

One should not seek to discover *the* process model. Process models are just a *view on reality*. Whether a process model is suitable or not, ultimately depends on the questions one would like to answer. Real-life processes are complex and may have many dimensions; models only provide a view on this reality. As discussed in Sect. 5.5, this means that the "3-D reality needs to be flattened into a 2-D process model" in order to apply process mining techniques. For instance, there are many "2-D slices" that one could take of a data set involving customer orders, order lines, deliveries, payments, replenishment orders, etc. Obviously, the different slices result in the discovery of different process models. Using the metaphor of a "process view", a discovered process model views reality from a particular "angle", is "framed", and is shown using a particular "resolution":

- A discovered model views reality from a particular *angle*. For example, the same process may be analyzed from the viewpoint of a complete order, a delivery, a customer, or an order line.
- A discovered model *frames* reality. The frame determines the boundaries of the process and selects the perspectives of interest (control-flow, information, resources, etc.).
- A discovered model provides a view at a specific *resolution*. The same process can be viewed using a coarser or finer granularity showing less or more details.

Figure 6.25 illustrates the "process view" metaphor. Given a data set it is possible to *zoom in*, i.e., selecting a smaller frame and increasing resolution, resulting in a more fine-grained model of a selected part of the process. It is also possible to *zoom out*, i.e., selecting a larger frame and decreasing resolution, resulting in a more coarse-grained model covering a larger part of the end-to-end process. Both the data set used as input and the questions that need to be answered determine which 2-D slices are most useful.

http://www.springer.com/978-3-662-49850-7

Process Mining
Data Science in Action
van der Aalst, W.M.P.
2016, XIX, 467 p. 250 illus., 13 illus. in color., Hardcover
ISBN: 978-3-662-49850-7