Chapter 7
Advanced Process Discovery Techniques

The α-algorithm nicely illustrates some of the main ideas behind process discovery. However, this simple algorithm is unable to manage the trade-offs involving the four quality dimensions described in Chap. 6 (fitness, simplicity, precision, and generalization). To successfully apply process mining in practice, one needs to deal with noise and incompleteness. This chapter focuses on more advanced process discovery techniques. The goal is not to present one particular technique in detail, but to provide an overview of the most relevant approaches. This will assist the reader in selecting the appropriate process discovery technique. Moreover, insights into the strengths and weaknesses of the various approaches support the correct interpretation and effective use of the discovered models.

7.1 Overview

Figure 7.1 summarizes the problems mentioned in the context of the α-algorithm. Each back dot represents a trace (i.e., a sequence of activities) corresponding to one or more cases in the event log. (Recall that multiple cases may have the same corresponding trace.) An event log typically contains only a fraction of the possible behavior, i.e., the dots should only be seen as *samples* of a much larger set of possible behaviors. Moreover, one is typically primarily interested in frequent behavior and not in all possible behavior, i.e., one wants to abstract from noise and therefore not all dots need to be relevant for the process model to be constructed.

Recall that we defined noise as infrequent or exceptional behavior. It is interesting to analyze such noisy behaviors, however, when constructing the overall process model, the inclusion of infrequent or exceptional behavior leads to complex diagrams. Moreover, it is typically impossible to make reliable statements about noisy behavior given the small set of observations. Figure 7.1 distinguishes between frequent behavior (solid rectangle with rounded corners) and all behavior (dashed rectangle), i.e., normal and noisy behavior. The difference between normal and noisy behavior is a matter of definition, e.g., normal behavior could be defined as the 80%

W. van der Aalst, *Process Mining*, DOI 10.1007/978-3-662-49851-4_7

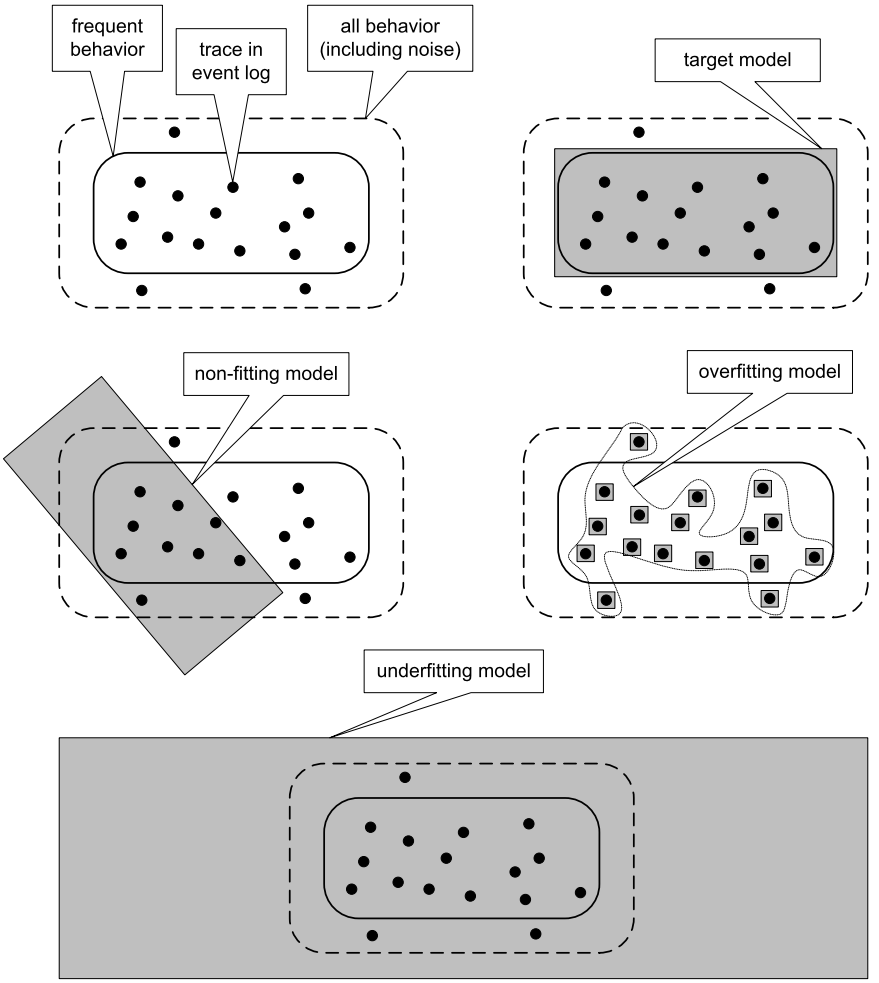

Fig. 7.1 Overview of the challenges that process discovery techniques need to address

most frequently occurring traces. Earlier we mentioned the *80/20 model*, i.e., the process model that is able to describe 80% of the behavior seen in the log. This model is typically relatively simple because the remaining 20% of the log may easily account for 80% of the variability in the process.

Let us assume that the two rectangles with rounded corners can be determined by observing the process infinitely long and that the process does not change (i.e., no concept drift). Based on these assumptions, Fig. 7.1 sketches four discovered models depicted by shaded rectangles. These discovered models are based on the example traces in the log, i.e., the black dots. The "ideal process model" allows for the behavior coinciding with the frequent behavior seen when the process would be observed ad infinitum while being in steady state. The "non-fitting model" in

Fig. 7.1 is unable to characterize the process well as it is not even able to capture the examples in the event log used to learn the model. The "overfitting model" does not generalize and only says something about the examples in the current event log. New examples will most likely not fit into this model. The "underfitting model" lacks precision and allows for behavior that would never be seen if the process would be observed ad infinitum.

Figure 7.1 illustrates the challenges process discovery techniques need to address: How to extract a simple target model that is not underfitting, overfitting, or non-fitting? Clearly, the α-algorithm is unable to do so. Therefore, we present more advanced approaches. However, before doing so, we describe typical characteristics of process discovery algorithms.

7.1.1 Characteristic 1: Representational Bias

The first, and probably most important, characteristic of a process discovery algorithm is its *representational bias*, i.e., the class of process models that can be discovered. For instance, the α-algorithm is only able to discover Petri nets in which each transition has a unique and visible label. Instead of Petri nets, some other representation can be used, e.g., a subclass of BPMN, EPCs, YAWL, hidden Markov models, transition systems, and causal nets. The representational bias determines the search space and potentially limits the expressiveness of the discovered model. Consider, for example, the three process models for event log $L_{11} = [\langle a, b, c\rangle^{20}, \langle a, c\rangle^{30}]$ in Fig. 6.21. If the representational bias allows for duplicate labels (two transitions with the same label) or silent (τ) transitions, a suitable WF-net can be discovered. However, if the representational bias does not allow for this, the discovery algorithm is destined to fail and will not find a suitable WF-net. The *workflow patterns* [155, 191] are a tool to discuss and identify the representational bias of a language. Here, we do not discuss the more than 40 control-flow patterns. Instead, we mention some typical representational limitations imposed by process discovery algorithms:

- *Inability to represent concurrency.* Low-level models, such as Markov models, flow charts, and transition systems, do not allow for the modeling of concurrency other than enumerating all possible interleavings. Recall that such a low-level model will need to show $2^{10} = 1024$ states and $10 \times 2^{10-1} = 5120$ transitions to model a process with 10 parallel activities. Higher level models (like Petri nets and BPMN) only need to depict 10 activities and $2 \times 10 = 20$ "local" states (states before and after each activity).
- *Inability to deal with (arbitrary) loops.* Many process discovery algorithms impose some limitations on loops, e.g., the α-algorithm needs a pre- and post-processing step to deal with shorts loops (see Figs. 6.11 and 6.13). The "Arbitrary Cycles" pattern [155, 191] is typically not supported by algorithms that assume the underlying model to be block-structured.

- *Inability to represent silent actions.* In some notations, it is impossible to model silent actions like the skipping of an activity. Although such events are not explicitly recorded in the event log, they need to be reflected in the model. This limits the expressive power as illustrated by Fig. 6.21.
- *Inability to represent duplicate actions.* In many notations there cannot be two activities having the same label. If the same activity appears in different parts of the process, but these different instances of the same activity cannot be distinguished in the event log, then most algorithms will assume a single activity thus creating causal dependencies (e.g., non-existing loops) that do not exist in the actual process.
- *Inability to model OR-splits/joins.* As shown in Chap. 3, YAWL, BPMN, EPCs, causal nets, etc. allow for the modeling of OR-splits and OR-joins; see for example the models depicted in Figs. 3.6, 3.10 and 3.13 using such constructs. If the representational bias of a discovery algorithm does not allow for OR-splits and OR-joins, then the discovered model may be more complex or the algorithm is unable to find a suitable model.
- *Inability to represent non-free-choice behavior.* Most algorithms do not allow for non-free-choice constructs, i.e., constructs where concurrency and choice meet. Figure 6.1 uses a non-free-choice construct, because places $p1$ and $p2$ serve both as an XOR-split (to choose between doing just e or both b and c) and as an AND-split (to start the concurrent activities b and c). This WF-net can be discovered by the α-algorithm. However, non-free-choice constructs can also represent non-local dependencies as is illustrated by the WF-net in Fig. 6.14. Such WF-nets cannot be discovered by the basic α-algorithm. Whereas WF-nets can express non-free-choice behavior, many discovery algorithms use a representation that cannot do so.
- *Inability to represent hierarchy.* Most process discovery algorithms work on "flat" models. A notable exception is the Fuzzy Miner [66] that extracts hierarchical models. Activities that have a lower frequency but that are closely related to other low frequent activities are grouped into subprocesses. The representational bias determines whether, in principle, hierarchical models can be discovered or not.

7.1.2 Characteristic 2: Ability to Deal With Noise

Noisy behavior, i.e., exceptional/infrequent behavior, should not be included in the discovered model (see Sect. 6.4.2). First of all, users typically want to see the mainstream behavior. Second, it is impossible to infer meaningful information on activities or patterns that are extremely rare. Therefore, the more mature algorithms address this issue by abstracting from exceptional/infrequent behavior. Noise can be removed by preprocessing the log, or the discovery algorithm can abstract from noise while constructing the model. The ability or inability to deal with noise is an important characteristic of a process discovery algorithm.

7.1.3 Characteristic 3: Completeness Notion Assumed

Related to noise is the issue of *completeness*. Most process discovery algorithms make an implicit or explicit completeness assumption. For example, the α-algorithm assumes that the relation $>_L$ is complete, i.e., if one activity can be directly followed by another activity, then this should be seen at least once in the log. Other algorithms make other completeness assumptions. Some algorithms assume that the event log contains all possible traces, i.e., a very strong completeness assumption. This is very unrealistic and results in overfitting models. Algorithms that are characterized by a strong completeness assumption tend to overfit the log. A completeness assumption that is too weak tends to result in underfitting models.

7.1.4 Characteristic 4: Approach Used

There are many different approaches to do the actual discovery. It is impossible to give a complete overview. Moreover, several approaches are partially overlapping in terms of the techniques used. Nevertheless, we briefly discuss five characteristic families of approaches.

7.1.4.1 Direct Algorithmic Approaches

The first family of process discovery approaches extracts some *footprint* from the event log and uses this footprint to *directly* construct a process model. The α-algorithm [157] is an example of such an approach: relation $>_L$ is extracted from the log and based on this relation a Petri net is constructed. There are several variants of the α-algorithm [11, 185, 186] using a similar approach. Approaches using so-called "language-based regions" [19, 28, 170] infer places by converting the event log into a system of inequations. In this case, the system of inequations can be seen as the footprint used to construct the Petri net. See [174] for a survey of process mining approaches producing a Petri net. The approaches described in [66, 183, 184] also extract footprints from event logs. However, these approaches take frequencies into account to address issues related to noise and incompleteness.

7.1.4.2 Two-Phase Approaches

The second family of process discovery approaches uses a two-step approach in which first a "low-level model" (e.g., a transition system or Markov model) is constructed. In the second step the low-level model is converted into a "high-level model" that can express concurrency and other (more advanced) control-flow patterns. An example of such an approach is described in [165]. Here a transition system is extracted from the log using a customizable abstraction mechanism. Subsequently, the transition system is converted into a Petri net using so-called "state-based regions" [34]. The resulting model can be visualized as a Petri net, but can also

be converted into other notations (e.g. BPMN and EPCs). Similar approaches can be envisioned using hidden Markov models [9]. Using an Expectation-Maximization (EM) algorithm such as the Baum–Welch algorithm, the "most likely" Markov model can be derived from a log. Subsequently this model is converted into high-level model. A drawback of such approaches is that the representational bias cannot be exploited during discovery. Moreover, some of the mappings are "lossy", i.e., the process model needs to be slightly modified to fit the target language. These algorithms also tend to be rather slow compared to more direct algorithmic approaches.

7.1.4.3 Divide-and-Conquer Approaches

Rather than using a single pass through the event log, it is also possible to try and break the problem into smaller problems. The inductive miner [88] aims to split the event log recursively into sublogs. For example, if one group of activities is preceded by another group of activities, but never the other way around, then we may deduce that these groups are in a sequence relation. Subsequently, the event log is decomposed based on the two groups of activities. Next to the sequence relation, the inductive miner also detects choices, concurrency and loops. The sublogs are decomposed until they refer to single activity. The way that the log is decomposed provides a structured process model. Various inductive process discovery techniques have been developed for process trees (Sect. 3.2.8) [88–91].

7.1.4.4 Computational Intelligence Approaches

Techniques originating from the field of *computational intelligence* form the basis for the third family of process discovery approaches. Examples of techniques are ant colony optimization, genetic programming, genetic algorithms, simulated annealing, reinforcement learning, machine learning, neural networks, fuzzy sets, rough sets, and swarm intelligence. These techniques have in common that they use an evolutionary approach, i.e., the log is not directly converted into a model but uses an iterative procedure to mimic the process of natural evaluation. It is impossible to provide an overview of computational intelligence techniques here. Instead we refer to [9, 102] and use the *genetic process mining* approach described in [12] as an example. This approach starts with an initial population of individuals. Each individual corresponds to a randomly generated process model. For each individual a fitness value is computed describing how well the model fits with the log. Populations evolve by selecting the fittest individuals and generating new individuals using genetic operators such as crossover (combining parts of two individuals) and mutation (random modification of an individual). The fitness gradually increases from generation to generation. The process stops once an individual (i.e., model) of acceptable quality is found.

7.1.4.5 Partial Approaches

The approaches described thus far produce a complete end-to-end process model. It is also possible to focus on rules or frequent patterns. In Sect. 4.5.1, an approach for *mining of sequential patterns* was described [131]. This approach is similar to the discovery of association rules, however, now the order of events is taken into account. Another technique using an Apriori-like approach is the *discovery of frequent episodes* [94] described in Sect. 4.5.2. Here a sliding window is used to analyze how frequent an "episode" (i.e., a partial order) is appearing. Similar approaches exist to learn declarative (LTL-based) languages like *Declare* [162].

In the remainder, we discuss four approaches in more detail: heuristic mining (Sect. 7.2), genetic process mining (Sect. 7.3), region-based mining (Sect. 7.4), and inductive mining (Sect. 7.5). The chapter concludes with a historical perspective on process discovery going back to the classical work of Marc Gold, Anil Nerode, Alan Biermann, and others.

7.2 Heuristic Mining

Heuristic mining algorithms as described in [183, 184] use a representation similar to causal nets (see Sect. 3.2.7). Moreover, these algorithms take frequencies of events and sequences into account when constructing a process model. The basic idea is that infrequent paths should not be incorporated into the model. Both the representational bias provided by causal nets and the usage of frequencies makes the approach much more robust than most other approaches.

7.2.1 Causal Nets Revisited

In Sect. 3.2.7, we introduced the notion of causal nets, also referred to as C-nets. Figure 7.2 shows another example of a C-net. There is one start activity a representing the registration of an insurance claim. There is one end activity e that closes the case. Activity a has three output bindings: $\{b, c\}$, $\{d\}$ and $\{e\}$, indicating that after completing a, activities b and c are activated, d is activated, or e is activated. Recall that only valid sequences are considered (see Definition 3.11) when reasoning about the behavior of a C-net. A binding sequence is valid if the sequence (a) starts with start activity $a_i = a$, (b) ends with end activity $a_o = e$, (c) only removes obligations that are pending, and (d) ends without any pending obligations. Suppose that a occurs with output binding $\{b, c\}$. After executing $\langle(a, \emptyset, \{b, c\})\rangle$, there are two pending obligations: (a, b) and (a, c). This indicates that in the future b should occur with a in its input binding. Similarly, c should occur with a in its input binding. Executing b removes the obligation (a, b), but creates a new obligation (b, e), etc. An example of a valid sequence is $\langle(a, \emptyset, \{b, c\}), (b, \{a\}, \{e\}), (c, \{a\}, \{e\}), (e, \{b, c\}, \emptyset)\rangle$. At the end, there are no

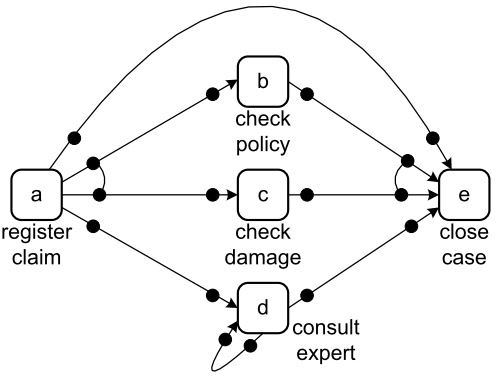

Fig. 7.2 Causal net modeling the handling of insurance claims

pending obligations. $\langle (a, \emptyset, \{d\}), (d, \{a\}, \{d\}), (d, \{d\}, \{e\}), (e, \{d\}, \emptyset)\rangle$ is another valid sequence. Because of the loop involving d there are infinitely many valid sequences.

The process modeled by Fig. 7.2 cannot be expressed as a WF-net (assuming that each transition has a unique visible label). This illustrates that C-nets are a more suitable representation for process discovery.

There are subtle differences between the notation used in [183, 184] and the C-nets used in this book. Whereas C-nets are very similar to the notation used in [183], there are relevant differences with [184]. In the original heuristic mining algorithm input and output bindings are a conjunction of mutually exclusive disjunctions, e.g., $O(t) = \{\{a, b\}, \{b, c\}, \{b, d\}\}$ means that t will activate a or b, and b or c, and b or d. These are exclusive or's. Hence, using the C-net semantics provided in Sect. 3.2.7 this corresponds to $O(t) = \{\{a, c, d\}, \{b\}\}$, i.e., either just b is activated or a, c and d are activated. C-nets are more intuitive and also more expressive (in a practical sense) than the original heuristic nets. Therefore, we use C-nets in the remainder.

7.2.2 Learning the Dependency Graph

To illustrate the basic concepts used by heuristic mining algorithms, we use the following event log:

$$L = \big[\langle a, e\rangle^5, \langle a, b, c, e\rangle^{10}, \langle a, c, b, e\rangle^{10}, \langle a, b, e\rangle^1, \langle a, c, e\rangle^1, \\ \langle a, d, e\rangle^{10}, \langle a, d, d, e\rangle^2, \langle a, d, d, d, e\rangle^1\big]$$

If we assume the three traces with frequency one to be noise, then the remaining 37 traces in the log correspond to valid sequences of the C-net in Fig. 7.2. Before explaining how to derive such a C-net, we first apply the α-algorithm to event log L. The result is shown in Fig. 7.3.

As expected, the α-algorithm does not infer a suitable model. The model does not allow for frequent traces, such as $\langle a, e\rangle$ and $\langle a, d, e\rangle$. By accident the model also

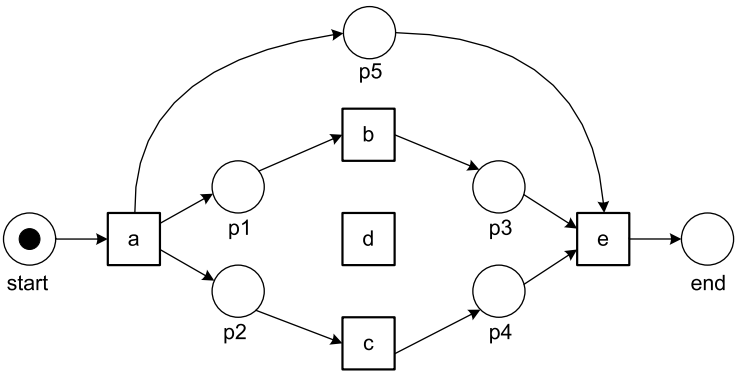

Fig. 7.3 WF-net constructed by the α-algorithm. The resulting model does not allow for $\langle a, e\rangle$, $\langle a, b, e\rangle$, $\langle a, c, e\rangle$, $\langle a, d, e\rangle$, $\langle a, d, d, e\rangle$, and $\langle a, d, d, d, e\rangle$

Table 7.1 Frequency of the "directly follows" relation in event log L: $|x >_L y|$ is the number of times x is directly followed by y in L

$\lvert >_L \rvert$	a	b	c	d	e
a	0	11	11	13	5
b	0	0	10	0	11
c	0	10	0	0	11
d	0	0	0	4	13
e	0	0	0	0	0

does not allow for infrequent traces such as $\langle a, b, e\rangle$, $\langle a, c, e\rangle$, and $\langle a, d, d, d, e\rangle$. There are two main problems. One problem is that the α-algorithm has a representational bias that does not allow for skipping activities (e.g., jumping from a to e) and cannot handle the requirement that d should be executed at least once when selected. The other problem is that the α-algorithm does not consider frequencies. Therefore, we use C-nets and take frequencies into account for heuristic mining.

Table 7.1 shows the number of times one activity is directly followed by another activity. For instance, $|d >_L d| = 4$, i.e., in the entire log d is followed four times by another d (two times in $\langle a, d, d, e\rangle$[2] and two times in $\langle a, d, d, d, e\rangle$[1]). Using Table 7.1 we can calculate the value of the *dependency relation* between any pair of activities.

Definition 7.1 (Dependency measure) Let L be an event log[1] over $\mathscr{A}$ and $a, b \in \mathscr{A}$. $|a >_L b|$ is the number of times a is directly followed by b in L, i.e.,

$$|a >_L b| = \sum_{\sigma \in L} L(\sigma) \times \left|\left\{1 \leq i < |\sigma| \;\middle|\; \sigma(i) = a \;\wedge\; \sigma(i+1) = b\right\}\right|$$

[1] Note that in this chapter we again assume that the event log is simple (like in Chap. 6) because at this stage we still abstract from the other perspectives.

Table 7.2 Dependency measures between the five activities based on event log L

$\lvert\Rightarrow_L\rvert$	a	b	c	d	e
a	$\frac{0}{0+1} = 0$	$\frac{11-0}{11+0+1} = 0.92$	$\frac{11-0}{11+0+1} = 0.92$	$\frac{13-0}{13+0+1} = 0.93$	$\frac{5-0}{5+0+1} = 0.83$
b	$\frac{0-11}{0+11+1} = -0.92$	$\frac{0}{0+1} = 0$	$\frac{10-10}{10+10+1} = 0$	$\frac{0-0}{0+0+1} = 0$	$\frac{11-0}{11+0+1} = 0.92$
c	$\frac{0-11}{0+11+1} = -0.92$	$\frac{10-10}{10+10+1} = 0$	$\frac{0}{0+1} = 0$	$\frac{0-0}{0+0+1} = 0$	$\frac{11-0}{11+0+1} = 0.92$
d	$\frac{0-13}{0+13+1} = -0.93$	$\frac{0-0}{0+0+1} = 0$	$\frac{0-0}{0+0+1} = 0$	$\frac{4}{4+1} = 0.80$	$\frac{13-0}{13+0+1} = 0.93$
e	$\frac{0-5}{0+5+1} = -0.83$	$\frac{0-11}{0+11+1} = -0.92$	$\frac{0-11}{0+11+1} = -0.92$	$\frac{0-13}{0+13+1} = -0.93$	$\frac{0}{0+1} = 0$

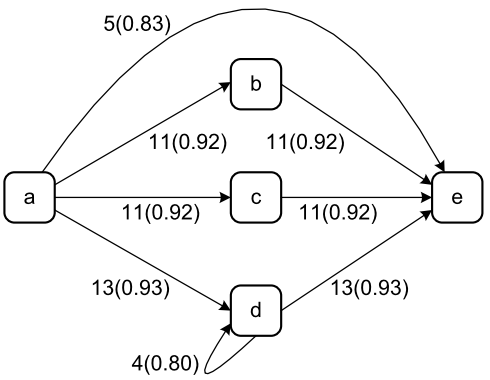

Fig. 7.4 Dependency graph using a threshold of 2 for $\lvert>_L\rvert$ and 0.7 for $\lvert\Rightarrow_L\rvert$: *each arc* shows the $\lvert>_L\rvert$ value and the $\lvert\Rightarrow_L\rvert$ value between brackets. For example, $\lvert a >_L d\rvert = 13$ and $\lvert a \Rightarrow_L d\rvert = 0.93$

$\lvert a \Rightarrow_L b\rvert$ is the value of the dependency relation between a and b:

$$\lvert a \Rightarrow_L b\rvert = \begin{cases} \frac{\lvert a>_L b\rvert - \lvert b>_L a\rvert}{\lvert a>_L b\rvert + \lvert b>_L a\rvert + 1} & \text{if } a \neq b \\ \frac{\lvert a>_L a\rvert}{\lvert a>_L a\rvert + 1} & \text{if } a = b \end{cases}$$

$\lvert a \Rightarrow_L b\rvert$ produces a value between -1 and 1. If $\lvert a \Rightarrow_L b\rvert$ is close to 1, then there is a strong positive dependency between a and b, i.e., a is often the cause of b. A value close to 1 can only be reached if a is often directly followed by b but b is hardly ever directly followed by a. If $\lvert a \Rightarrow_L b\rvert$ is close to -1, then there is a strong negative dependency between a and b, i.e., b is often the cause of a. There is a special case for $\lvert a \Rightarrow_L a\rvert$. If a is often followed by a this suggests a loop and a strong reflexive dependency. However, $\frac{\lvert a>_L a\rvert - \lvert a>_L a\rvert}{\lvert a>_L a\rvert + \lvert a>_L a\rvert + 1} = 0$ by definition. Therefore, the following formula is used: $\lvert a \Rightarrow_L a\rvert = \frac{\lvert a>_L a\rvert}{\lvert a>_L a\rvert + 1}$. Table 7.2 shows the dependency measures for event log L.

Using the information in Tables 7.1 and 7.2 we can derive the so-called *dependency graph*. The dependency graph corresponds to the dependency relation $D \subseteq A \times A$ in Definition 3.8. In a dependency graph only arcs are shown that meet certain *thresholds*. The dependency graph shown in Fig. 7.4 uses a threshold of 2 for $\lvert>_L\rvert$ and 0.7 for $\lvert\Rightarrow_L\rvert$, i.e., an arc between x and y is only included if $\lvert x >_L y\rvert \geq 2$ and $\lvert x \Rightarrow_L y\rvert \geq 0.7$.

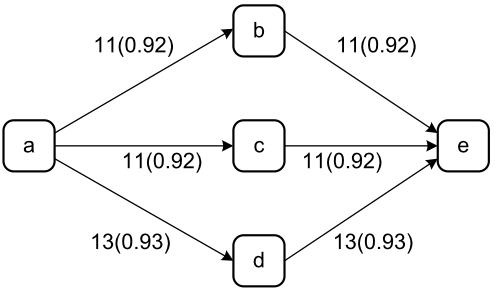

Fig. 7.5 Dependency graph using a threshold of 5 for $|>_L|$ and 0.9 for $|\Rightarrow_L|$. The self loop involving d disappeared because $|d >_L d| = 4 < 5$ and $|d \Rightarrow_L d| = 0.80 < 0.9$. The connection between a and e disappeared because $|a \Rightarrow_L e| = 0.83 < 0.9$

Figure 7.5 shows another dependency graph based on Tables 7.1 and 7.2 using higher thresholds. As a result two arcs disappear. Obviously, the dependency graph does not show the routing logic, e.g., one cannot see that after a, both b and c can be executed concurrently. Nevertheless, the dependency graph reveals the "backbone" of the process model.

The two dependency graphs show that, for a given event log, different models can be generated by adjusting the thresholds. This way the user can decide to focus on the mainstream behavior or to also include low frequent (i.e., noisy) behavior. In Figs. 7.4 and 7.5 the set of activities is the same. The two thresholds cannot be used to remove low frequent activities. This should be done by preprocessing the event log. For example, one could decide to concentrate on the most frequent activities and simply remove all other activities from the event log before calculating the dependency measures. Other techniques such as the one used by the Fuzzy Miner [66] remove such activities while realizing the dependency graph.

As shown in [183, 184], various refinements can be used to improve the dependency graph. For instance, it is possible to better deal with loops of length two and long distance dependencies. (See discussion in context of the processes shown in Figs. 6.13 and 6.14.)

7.2.3 Learning Splits and Joins

The goal of heuristic mining is to extract a C-net $C = (A, a_i, a_o, D, I, O)$ from the event log. The nodes of the dependency graph correspond to the set of activities A. The arcs of the dependency graph correspond to the dependency relation D. In a C-net, there is a unique start activity a_i and a unique end activity a_o. This is just a technicality. One can preprocess the log and insert artificial start and end events to each trace. Hence the assumption that there is a unique start activity a_i and a unique end activity a_o imposes no practical limitations. In fact, it is convenient to have a clear start and end. We also assume that in the dependency graph all activities are on a path from a_i to a_o. Activities that are not on such a path should be removed or the thresholds need to be adjusted locally such that a minimal set of connections is established. It makes no sense to include activities that are not on a path from a_i

to a_o: such an activity would be dead or could be active before the case starts, and does not contribute to the completion of the case. Therefore, we can assume that, by constructing the dependency graph, we already have the core structure of the C-net: (A, a_i, a_o, D). Hence, only the functions $I \in A \rightarrow AS$ and $O \in A \rightarrow AS$ need to be derived to complete the C-net.

Given a dependency graph (A, a_i, a_o, D), we define $\circ a = \{a' \in A \mid (a', a) \in D\}$ and $a\circ = \{a' \in A \mid (a, a') \in D\}$ for any $a \in A$. Clearly, $I(a_i) = O(a_o) = \{\emptyset\}$. There are $2^{|\circ a|} - 1$ potential elements for $I(a)$ for any $a \neq a_i$ and $2^{|a\circ|} - 1$ potential elements for $O(a)$ for any $a \neq a_o$. Consider, for example, the dependency graph shown in Fig. 7.4. $a\circ = \{b, c, d, e\}$. Hence, $O(a)$ has $2^4 - 1 = 15$ potential output bindings: $\{b\}$, $\{c\}$, $\{d\}$, $\{e\}$, $\{b, c\}$, $\{b, d\}$, …, $\{b, c, d, e\}$. $O(b)$ has only $2^1 - 1 = 1$ possible element, $\{e\}$. $I(b)$ also has just one possible element, $\{a\}$. $O(d)$ has $2^2 - 1 = 3$ potential output bindings: $\{d\}$, $\{e\}$, and $\{d, e\}$. $I(d)$ also has $2^2 - 1 = 3$ potential input bindings, $\{a\}$, $\{d\}$, and $\{a, d\}$.

If there is just one potential binding element, then this element should be taken. Hence, $I(b) = \{\{a\}\}$, $I(c) = \{\{a\}\}$, $O(b) = \{\{e\}\}$, and $O(c) = \{\{e\}\}$. For the other input and output bindings, subsets need to be selected based on the event log. To do this, the event log is replayed on the dependency graph to see how frequent output sets are triggered.

Consider, for example, $O(d)$. In event log L, activity d is four times followed by just d and 13 times by just e; d is never followed by both d and e. Therefore, $\{e\}$ is definitely included in $O(d)$ because it is the most frequent output binding. $\{d\}$ may be included depending on the threshold for including bindings. If we assume that both possible bindings are included, then $O(d) = \{\{d\}, \{e\}\}$. Similarly, we find $I(d) = \{\{a\}, \{d\}\}$. Let us now consider $O(a)$. As indicated earlier there are $2^4 - 1 = 15$ possible output bindings. Replaying the event log on the dependency graph shows that a is 5 times followed by e (in traces $\langle a, e\rangle^5$), a is 20 times followed by both b and c (in traces $\langle a, b, c, e\rangle^{10}$ and $\langle a, c, b, e\rangle^{10}$), and a is 13 times followed by d (in traces $\langle a, d, e\rangle^{10}$, $\langle a, d, d, e\rangle^2$, and $\langle a, d, d, d, e\rangle^1$). Activity a is once followed by just b (in trace $\langle a, b, e\rangle$) and is once followed by just c (in trace $\langle a, c, e\rangle$). Let us assume that the latter two output bindings are below a preset threshold. Then $O(a) = \{\{b, c\}, \{d\}, \{e\}\}$, i.e., of the 15 possible output bindings only three are frequent enough to be included.

Many replay strategies are possible to determine the frequency of a binding. In [119, 183, 184] heuristics are used to select the bindings to be included. In [4], a variant of the A^* algorithm is used to find an "optimal" replay of traces on the dependency graph. The semantics of a C-net are global, i.e., the validity of a binding sequence cannot be determined locally (like in a Petri net). We refer to [4, 119, 183, 184] for example replay strategies.

By replaying the event log on the dependency graph, we can estimate the frequencies of input and output bindings. Using thresholds, it is possible to exclude bindings based on their frequencies. This results in functions I and O, thus completing the C-net. Figure 7.6 shows the C-net based on the dependency graph in Fig. 7.4. As shown $O(a) = \{\{b, c\}, \{d\}, \{e\}\}$ and $I(e) = \{\{a\}, \{b, c\}, \{d\}\}$. Bindings $\{b\}$ and $\{c\}$ are not included in $O(a)$ and $I(e)$ because they occur only once (below

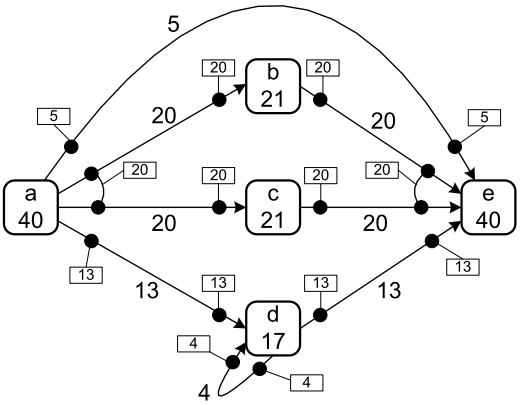

Fig. 7.6 C-net derived from the event log L. *Each node* shows the frequency of the corresponding activity. *Every arc* has a frequency showing how often both activities agreed on a common binding. The frequencies of input and output bindings are also depicted, e.g., 20 of the 40 occurrences of a were followed by the concurrent execution of b and c

threshold). Figure 7.6 also shows the frequencies of activities, dependencies, and bindings. For example, activity a occurred 40 times. The output binding $\{b, c\}$ of a occurred 20 times. Activity d occurred 17 times: 13 times triggered by a and 4 times by d itself. Activity b occurred 21 times. The frequency of the only input binding $\{a\}$ is only 20. This difference is caused by the exclusion of the infrequent output binding $\{b\}$ of a (this binding occurs only in trace $\langle a, b, e\rangle$). A similar difference can be found for activity c.

Figure 7.7 provides a more intuitive visualization of the C-net of Fig. 7.6. Now the thickness of the arcs corresponds to the frequencies of the corresponding paths. Such visualizations are important to get insight into the main process flows. In Chap. 15, we will adopt the metaphor of a roadmap to visualize process models. A roadmap highlights highways using thick lines and bright colors. At the same time insignificant roads are not shown. Figure 7.7 illustrates that the same can be done using heuristic mining.

The approach presented in this section is quite generic and can be applied to other representations. A notable example is the *fuzzy mining* approach described in [65, 66]. This approach provides an extensible set of parameters to determine which activities and arcs need to be included. Moreover, the approach can construct hierarchical models, i.e., less frequent activities may be moved to subprocesses. Also the metaphor of a roadmap is exploited to create process models that can be understood easily while providing information on the frequency and importance of activities and paths (cf. Sect. 15.1.3).

7.3 Genetic Process Mining

The α-algorithm and techniques for heuristic and fuzzy mining provide process models in a direct and deterministic manner. *Evolutionary approaches* use an iterative procedure to mimic the process of natural evolution. Such approaches are not deterministic and depend on randomization to find new alternatives. This sec-

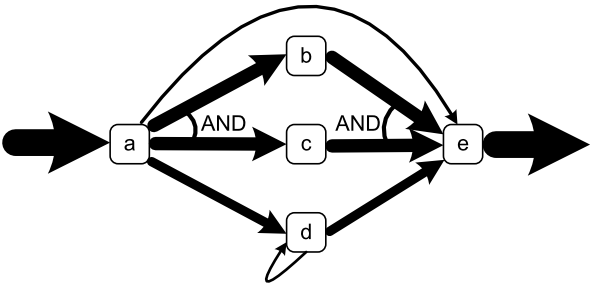

Fig. 7.7 Alternative visualization of the C-net clearly showing the "highways" in the process model

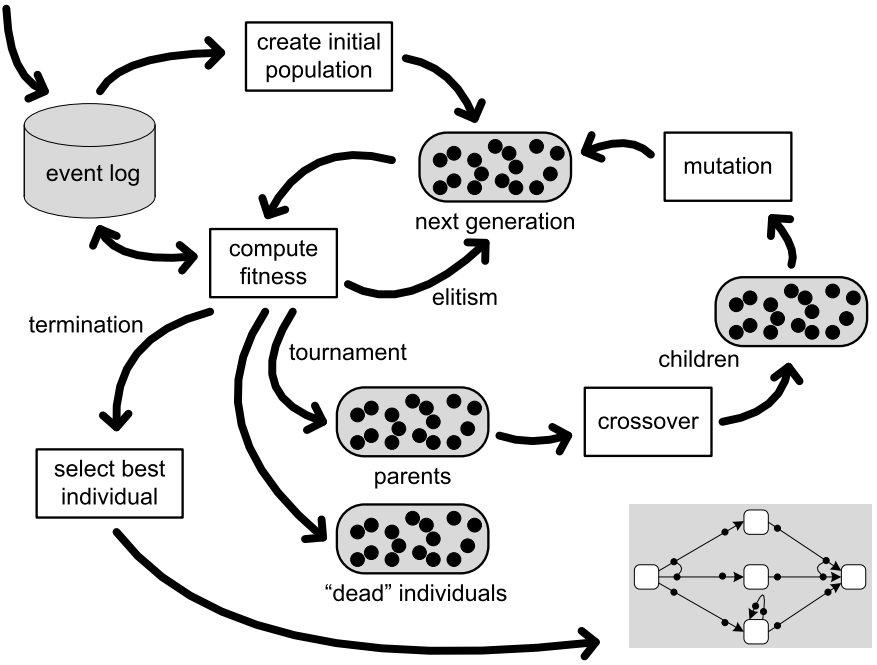

Fig. 7.8 Overview of the approach used for genetic process mining

tion describes *genetic process mining* [12] as an example of a process discovery approach using a technique from the field of computational intelligence.

Figure 7.8 shows an overview of the approach used in [12]. Like in any genetic algorithm there are four main steps: (a) initialization, (b) selection, (c) reproduction, and (d) termination.

In the *initialization* step the initial population is created. This is the first generation of individuals to be used. Here an individual is a process model. Using the activity names appearing in the log, process models are created *randomly*. There may

be hundreds or even thousands individuals in each generation. The process models (i.e., individuals) in the initial population may have little to do with the event log; the activity names are the same but the behaviors of the initial models are likely to be very different from the behavior seen in the event log. However, "by accident" the generated models may have parts that fit parts of the event log due random effects and the large number of individuals.

In the *selection* step, the fitness of each individual is computed. A fitness function determines the quality of the individual in relation to the log.[2] In Sect. 6.4.3, we discussed different ways of measuring the quality of a model. A simple criterion is the proportion of traces in the log that can be replayed by the model. This is not a good fitness function, because it is very likely that none of the models in the initial population can replay any of the traces in the event log. Moreover, using this criterion an over-general model like the "flower model" would have a high fitness. Therefore, a more refined fitness function needs to be used that also rewards the partial correctness of the model and takes into account all four competing quality criteria described in Sect. 6.4.3. The best individuals, i.e., the process models having the highest fitness value are moved to the next generation. This is called *elitism*. For instance, the best 1% of the current generation is passed on to the next generation without any modifications. Through *tournaments* "parents" are selected for creating new individuals. Tournaments among individuals and elitism should make sure that the "genetic material of the best process models" has the highest probability of being used for the next generation: *survival of the fittest*. As a result, individuals with a poor fitness are unlikely to survive. Figure 7.8 refers to such models as "dead" individuals.

In the *reproduction* phase the selected parent individuals are used to create new offspring. Here two genetic operators are used: *crossover* and *mutation*. For crossover two individuals are taken and used to create two new models; these end up in the pool with "child models" shown in Fig. 7.8. These child models share parts of the genetic material of their parents. The resulting children are then modified using mutation, e.g., randomly adding or deleting a causal dependency. Mutation is used to insert new generic material in the next generation. Without mutation, evolution beyond the genetic material in the initial population is impossible.

Through reproduction (i.e., crossover and mutation) and elitism a new generation is created. For the models in this generation the fitness is computed. Again the best individuals move on to the next round (elitism) or are used to produce new offspring. This is repeated and the expectation is that the "quality" of each generation gets better and better. The evolution process *terminates* when a satisfactory solution is found, i.e., a model having at least the desired fitness. Depending on the event log it may take a very long time to converge. In fact, due to the representational bias and noise in the event log there may not be a model that has the desired level of

[2]Note that we overload the term "fitness" in this book. On the one hand, we use it to refer to the ability to replay the event log (see Sects. 6.4.3 and 8.2). On the other hand, we use it for the selection of individuals in genetic process mining. Note that the latter interpretation includes the former, but also adds other elements of the four criteria mentioned in Sect. 6.4.3.

fitness. Therefore, other termination criteria may be added (e.g., a maximum number of generations or stopping when 10 successive generations do not produce better individuals). When terminating, a model with the best fitness is returned.

The approach described in Fig. 7.8 is very general. When actually implementing a genetic process mining algorithm the following design choices need to made:

- *Representation of individuals.* Each individual corresponds to a process model described in a particular language, e.g., Petri nets, C-nets, BPMN, or EPCs. This choice is important as it determines the class of processes that can be discovered (representational bias). Moreover, it should be possible to define suitable genetic operators for the representation chosen. In [12], a variant of C-nets is used.
- *Initialization.* For the initial population, models need to be generated randomly. In [12], two approaches are proposed: (a) an approach where with a certain probability a causal dependency between two activities is inserted to create C-nets and (b) an approach in which a randomized variant of heuristic mining is used to create an initial population with a higher average fitness than purely randomly generated C-nets.
- *Fitness function.* Here, the challenge is to define a function that balances the four competing quality criteria described in Sect. 6.4.3. Many fitness functions can be defined. The fitness function drives the evolution process and can be used to favor particular models. In [12], the proportion of events in the log that can be parsed by the model is computed. This is combined with penalties for having many enabled activities (cf. the flower model in Fig. 6.23).
- *Selection strategy (tournament and elitism).* The genetic algorithm needs to determine the fraction of individuals that go to the next round without any changes. Through elitism it is ensured that good models do not get lost due to crossover or mutation. There are different approaches to select parents for crossover. In [12], parents are selected by randomly taking five individuals and then selecting the best one, i.e., a tournament among five randomly selected models is used.
- *Crossover.* The goal of crossover is to recombine existing genetic material. The basic idea is to create a new process model that uses parts of its two parent models. In [10, 12], both parents are C-nets having the same set of activities. One of these common activities is selected randomly, say a. Let $I_1(a)$ and $O_1(a)$ be the possible bindings of one parent, and let $I_2(a)$ and $O_2(a)$ be the potential bindings of the other parent. Now parts of $I_1(a)$ are swapped with parts of $I_2(a)$ and parts of $O_1(a)$ are swapped with parts of $O_2(a)$. Subsequently, both C-nets are repaired as bindings need to be consistent among activities. The crossover of two parent models results in two new child models. These child models may be mutated before being added to the next generation.
- *Mutation.* The goal of mutation is to randomly insert new genetic material. In [10, 12], each activity in each child resulting from crossover has a small probability of being selected for mutation. If this is the case, say a is selected for mutation, then $I(a)$ or $O(a)$ is randomly modified by adding or removing potential bindings.

The above list shows that many design decisions need to be taken when developing a genetic process mining algorithm. We refer to [10, 12] for concrete examples.

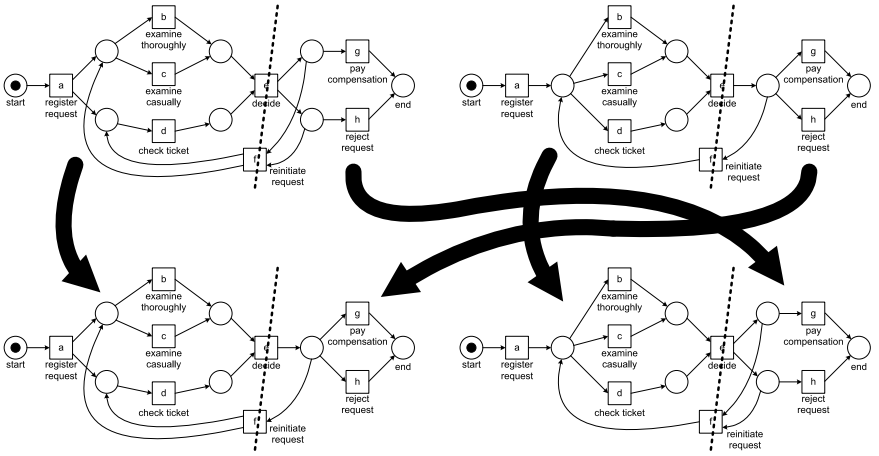

Fig. 7.9 Two parent models (*top*) and two child models resulting from a crossover. The crossover points are indicated by the *dashed lines*

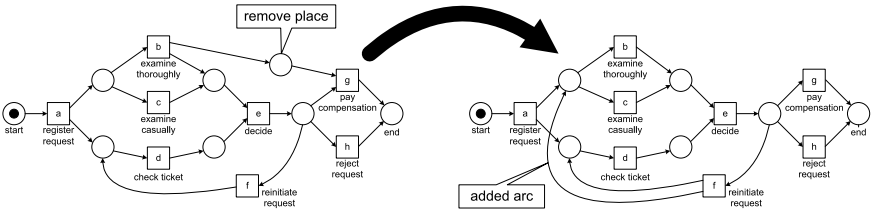

Fig. 7.10 Mutation: a place is removed and an arc is added

An essential choice is the representation of individuals. The approach described in [10, 12] uses a variant of C-nets similar to the notation used for the initial heuristic mining algorithm [184]. However, many other representations are possible.

To illustrate the genetic operators, we show a crossover example and a mutation example. For clarity we use Petri nets to describe the individuals before and after modification. Figure 7.9 shows two "parent" models and two "child" models resulting from crossover. In this example, the crossover point is the line through activities e and f. Figure 7.10 shows an example of mutation: one place is removed and one arc is added.

Figures 7.9 and 7.10 nicely illustrate the idea behind the two genetic operators: crossover and mutation. However, the realization of such operators is not as simple as these examples suggest. Typically repair actions are needed after crossover and mutation. For instance, the resulting model may no longer be a WF-net or C-net. Again we refer to [10, 12] for concrete examples.

Genetic process mining is *flexible* and *robust*. Like heuristic mining techniques, it can deal with noise and incompleteness. The approach can also be adapted and extended easily. By changing the fitness function it is possible to give preference to

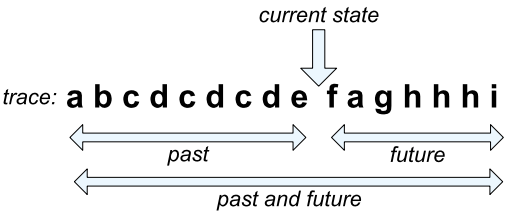

Fig. 7.11 Every position in a trace corresponds to a state, e.g., the state after executing the first nine events of a trace consisting of 16 events. To characterize the state, the past and/or future can be used as "ingredients"

particular constructs. Unfortunately, like most evolutionary approaches, genetic process mining is *not very efficient* for larger models and logs. It may take a very long time to discover a model having an acceptable fitness. In theory, it can be shown that suitably chosen genetic operators guarantee that eventually a model with optimal fitness will be produced. However, in practice this argument is not useful given the potentially excessive computation times. It is also useful to combine heuristics with genetic process mining. In this case, genetic process mining is used to improve a process model obtained using heuristic mining. This saves computation time and may result in models that could never have been obtained through conventional algorithms searching only for local dependencies.

7.4 Region-Based Mining

In the context of Petri nets, researchers have been looking at the so-called *synthesis problem*, i.e., constructing a system model from a description of its behavior. State-based regions can be used to construct a Petri net from a transition system. Language-based regions can be used to construct a Petri net from a prefix-closed language. Synthesis approaches using language-based regions can be applied directly to event logs. To apply state-based regions, one first needs to create a transition system.

7.4.1 Learning Transition Systems

To construct a Petri net using state-based regions, we first need to discover a transition system based on the traces in the event log. Recall that a transition system can be described by a triplet $TS = (S, A, T)$ where S is the set of *states*, $A \subseteq \mathscr{A}$ is the set of *activities*, and $T \subseteq S \times A \times S$ is the set of *transitions*. $S^{start} \subseteq S$ is the set of *initial states*. $S^{end} \subseteq S$ is the set of *final states*. (See Sect. 3.2.1 for an introduction to transition systems.)

How to construct $TS = (S, A, T)$ based on some simple event log L over $\mathscr{A}$, i.e., $L \in \mathbb{B}(\mathscr{A}^*)$? An obvious choice is to take A to be the set of activities in the simple event log. In order to determine the set of states, each "position" in each trace in the log needs to be mapped onto a corresponding state. This is illustrated by Fig. 7.11.

Let $\sigma' = \langle a, b, c, d, c, d, c, d, e, f, a, g, h, h, h, i \rangle \in L$ be a trace in the event log. Every position in this trace, i.e., before the first event, in-between two events, or after the last event should correspond to a state in the transition system. Consider, for example, the state shown in Fig. 7.11. The partial trace $\sigma'_{past} = \langle a, b, c, d, c, d, c, d, e \rangle$ describes the past of the corresponding case. $\sigma'_{future} = \langle f, a, g, h, h, h, i \rangle$ describes the future of this case. A *state representation* function $l^{state}()$ is a function that, given some sequence σ and a number k indicating the number of events of σ that have occurred, produces some state, e.g., the set of activities that have occurred in the first k events.

Let $\sigma = \langle a_1, a_2, \ldots, a_n \rangle \in L$ be a trace of length n. $l_1^{state}(\sigma, k) = hd^k(\sigma) = \langle a_1, a_2, \ldots, a_k \rangle$ is an example of a state representation function. Recall that $hd^k(\sigma)$ was defined in Sect. 5.2; the function returns the "head" of the sequence σ consisting of the first k elements. $l_1^{state}(\sigma, k)$ describes the current state by the *full history* of the case after k events. For instance, $l_1^{state}(\sigma', 9) = \langle a, b, c, d, c, d, c, d, e \rangle$.

$l_2^{state}(\sigma, k) = tl^{n-k}(\sigma) = \langle a_{k+1}, a_{k+2}, \ldots, a_n \rangle$ is another example of a state representation function. $l_2^{state}(\sigma, k)$ describes the current state by the *full future* of the case after k events. $l_2^{state}(\sigma', 9) = \langle f, a, g, h, h, h, i \rangle$.

$l_3^{state}(\sigma, k) = \partial_{multiset}(hd^k(\sigma)) = [a_1, a_2, \ldots, a_k]$ is a state representation function converting the full history into a multi-set. This function assumes that for the current state the order of events is not important, only the frequency of activities matters. $l_3^{state}(\sigma', 9) = [a^1, b^1, c^3, d^3, e^1]$, i.e., in the state shown in Fig. 7.11 a, b, and e have been executed once and both c and d have been executed three times.

$l_4^{state}(\sigma, k) = \partial_{set}(hd^k(\sigma)) = \{a_1, a_2, \ldots, a_k\}$ is a state representation function taking a set representation of the full history. For this state representation function the order and frequency of activities do not matter. For the current state it only matters which activities have been executed at least once. $l_4^{state}(\sigma', 9) = \{a, b, c, d, e\}$.

Functions $l_1^{state}()$, $l_3^{state}()$, and $l_4^{state}()$ all consider the full history of the case after k events: $l_1^{state}()$ does not abstract from the order and frequency of past activities, $l_3^{state}()$ abstracts from the order, and $l_4^{state}()$ abstracts from both order and frequency. Hence, $l_4^{state}()$ provides a coarser abstraction than $l_1^{state}()$. By definition $l_4^{state}(\sigma_1, k) = l_4^{state}(\sigma_2, k)$ if $l_1^{state}(\sigma_1, k) = l_1^{state}(\sigma_2, k)$ (but not the other way around). Function $l_2^{state}()$ is based on the future rather than the past.

Using some state representation function $l^{state}()$ we can automatically construct a transition system based on some event log L.

Definition 7.2 (Transition system based on event log) Let $L \in \mathbb{B}(\mathscr{A}^*)$ be an event log and $l^{state}()$ a state representation function. $TS_{L,l^{state}()} = (S, A, T)$ is a transition system based on L and $l^{state}()$ with:

- $S = \{l^{state}(\sigma, k) \mid \sigma \in L \ \wedge \ 0 \leq k \leq |\sigma|\}$ is the state space;
- $A = \{\sigma(k) \mid \sigma \in L \ \wedge \ 1 \leq k \leq |\sigma|\}$ is the set of activities;
- $T = \{(l^{state}(\sigma, k), \sigma(k+1), l^{state}(\sigma, k+1)) \mid \sigma \in L \ \wedge \ 0 \leq k < |\sigma|\}$ is the set of transitions;
- $S^{start} = \{l^{state}(\sigma, 0) \mid \sigma \in L\}$ is the set of initial states; and
- $S^{end} = \{l^{state}(\sigma, |\sigma|) \mid \sigma \in L\}$ is the set of final states.

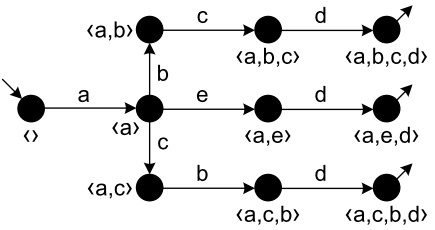

Fig. 7.12 Transition system $TS_{L_1,l_1^{state}()}$ derived from $L_1 = [\langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^2, \langle a,e,d\rangle]$ using $l_1^{state}(\sigma,k) = hd^k(\sigma)$

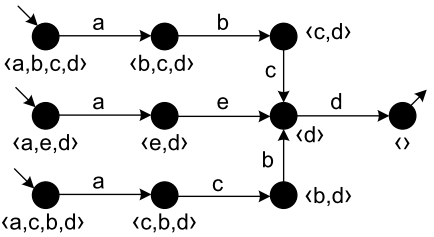

Fig. 7.13 Transition system $TS_{L_1,l_2^{state}()}$ derived from L_1 using $l_2^{state}(\sigma,k) = tl^{|\sigma|-k}(\sigma)$

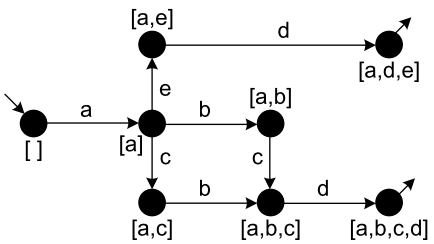

Fig. 7.14 Transition system $TS_{L_1,l_3^{state}()}$ derived from L_1 using $l_3^{state}(\sigma,k) = \partial_{multiset}(hd^k(\sigma))$

Let us consider event log $L_1 = [\langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^2, \langle a,e,d\rangle]$. Figure 7.12 shows transition system $TS_{L_1,l_1^{state}()}$. Consider, for example, a case with trace $\sigma = \langle a,b,c,d\rangle$. Initially, the case is in state $l_1^{state}(\sigma,0) = \langle\,\rangle$. After executing a the case is in state $l_1^{state}(\sigma,1) = \langle a\rangle$. After executing b state $l_1^{state}(\sigma,2) = \langle a,b\rangle$ is reached. Executing c results in state $l_1^{state}(\sigma,3) = \langle a,b,c\rangle$. Executing the last event d results in state $l_1^{state}(\sigma,4) = \langle a,b,c,d\rangle$. The five states visited by this case are added to the transition system. The corresponding transitions are also added. The same is done for $\langle a,c,b,d\rangle$ and $\langle a,e,d\rangle$, thus resulting in the transition system of Fig. 7.12.

Using state representation function $l_2^{state}()$ we obtain transition system $TS_{L_1,l_2^{state}()}$ shown in Fig. 7.13. In this transition system there are three initial states and only one final state, because this abstraction uses the future rather than the past. Consider, for example, a case with trace $\sigma = \langle a,e,d\rangle$. Initially, the case is in state $l_2^{state}(\sigma,0) = \langle a,e,d\rangle$, i.e., all three activities still need to occur. After executing a the case is in state $l_2^{state}(\sigma,1) = \langle e,d\rangle$. After executing e state $l_2^{state}(\sigma,2) = \langle d\rangle$ is reached. Executing the last event d results in state $l_2^{state}(\sigma,3) = \langle\,\rangle$.

Transition system $TS_{L_1,l_3^{state}()}$ is shown in Fig. 7.14. Here, the states are represented by the multi-sets of activities that have been executed before. For instance, $l_3^{state}(\langle a,b,c,d\rangle,3) = [a,b,c]$. Because there are no repeated activities $TS_{L_1,l_4^{state}()}$

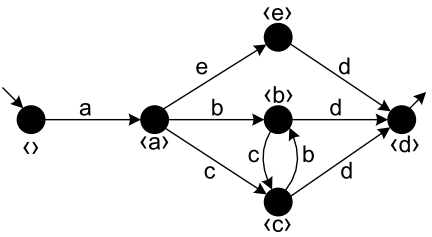

Fig. 7.15 Transition system $TS_{L_1, l_5^{state}()}$ derived from L_1 using $l_5^{state}(\sigma, k) = tl^1(hd^k(\sigma))$

is identical to $TS_{L_1, l_3^{state}()}$ apart from the naming of states, e.g., $l_4^{state}(\langle a, b, c, d\rangle, 3) = \{a, b, c\}$ rather than $[a, b, c]$.

The sets of traces allowed by the three transition systems shown in Figs. 7.12, 7.13, and 7.14 are the same: $\langle a, b, c, d\rangle$, $\langle a, c, b, d\rangle$, $\langle a, e, d\rangle$. This is not always the case. Add, for example, the trace $\langle a, c, b, f, f\rangle$ to L_1. In this case, $TS_{L_1, l_4^{state}()}$ allows for traces $\langle a, b, c, f, f\rangle$ and $\langle a, c, b, f, f, f, f, f\rangle$, i.e., b and c may be swapped and any number of f events is allowed at the end. $TS_{L_1, l_3^{state}()}$ allows for traces $\langle a, b, c, f, f\rangle$ and $\langle a, c, b, f, f\rangle$, but not $\langle a, c, b, f, f, f, f, f\rangle$. $TS_{L_1, l_1^{state}()}$ allows for trace $\langle a, c, b, f, f\rangle$, but not $\langle a, b, c, f, f\rangle$. Since $l_4^{state}()$ provides a coarser abstraction than $l_1^{state}()$, it generalizes more.

The state representation functions mentioned thus far are just examples. Depending on the desired abstraction, another state representation function can be defined. The essential question is whether partially executed cases are considered to be in the same state or not. For instance, if we assume that only the last activity matters, we can use state representation function $l_5^{state}(\sigma, k) = tl^1(hd^k(\sigma))$. This results in transition system $TS_{L_1, l_5^{state}()}$ shown in Fig. 7.15. Now, states in the transition system are labeled with the last activity executed. For the initial state this results in the empty sequence. $TS_{L_1, l_5^{state}()}$ allows for the traces in the event log, but also traces such as $\langle a, b, c, b, c, d\rangle$. Another example is $l_6^{state}(\sigma, k) = hd^3(tl^{|\sigma|-k}(\sigma))$, i.e., the state is determined by the next three events.

Thus far we only considered a simple event log as input. Real-life event logs contain much more information as was shown in Chap. 5 (cf. Definition 5.3 and the XES format). Information about resources and data can also be taken into account when constructing a transition system. This information can be used to identify states and to label transitions. For example, states may encode whether the customer being served is a gold or silver customer. Transitions can be labeled with resource names rather than activity names. See [165] for a systematic treatment of the topic.

A transition system defines a "low-level" process model. Unfortunately, such models cannot express higher level constructs and suffer from the "state explosion" problem. As indicated before, a simple process with 10 parallel activities already results in a transition system with $2^{10} = 1024$ states and $10 \times 2^{10-1} = 5120$ transitions. Fortunately, state-based regions can be used to synthesize a more compact model from such transition systems.

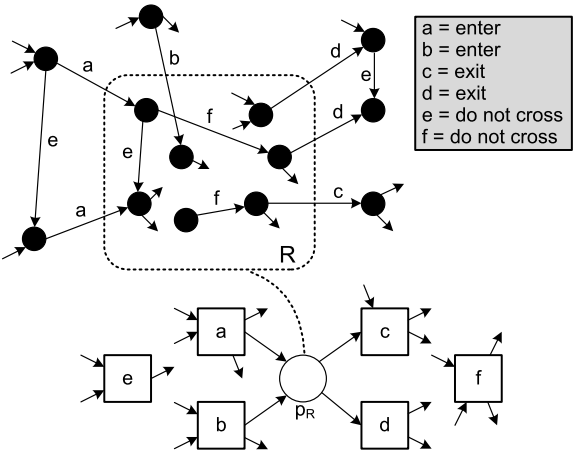

Fig. 7.16 Region R corresponding to place p_R. All activities can be classified into *entering* the region (a and b), *leaving* the region (c and d), and *non-crossing* (e and f)

7.4.2 Process Discovery Using State-Based Regions

After transforming an event log into a low-level transition system we can synthesize a Petri net from it. In turn, this Petri net can be used to construct a process model in some other high-level notation (e.g., BPMN, UML activity diagrams, YAWL, and EPCs). The challenge is to fold a large transition system into a smaller Petri net by detecting concurrency. The core idea is to discover *regions* that correspond to *places*. A region is a set of states such that all activities in the transition system "agree" on the region.

Definition 7.3 (State-based region) Let $TS = (S, A, T)$ be a transition system and $R \subseteq S$ be a subset of states. R is a *region* if for each activity $a \in A$ one of the following conditions hold:

1. All transitions $(s_1, a, s_2) \in T$ *enter* R, i.e., $s_1 \notin R$ and $s_2 \in R$;
2. All transitions $(s_1, a, s_2) \in T$ *exit* R, i.e., $s_1 \in R$ and $s_2 \notin R$; or
3. All transitions $(s_1, a, s_2) \in T$ *do not cross* R, i.e., $s_1, s_2 \in R$ or $s_1, s_2 \notin R$.

Let R be a region. In this case all activities can be classified into *entering* the region, *leaving* the region, and *non-crossing*. An activity cannot be entering the region in one part of the transition system and exiting the region in some other part. Figure 7.16 illustrates the concept. The dashed rectangle describes a region R, i.e., a set of states in the transition system. All activities need to take a position with respect to this region. All a-labeled transitions enter region R. If there would be a transition with an a label not connecting a state outside the region to a state inside the region, then R would not be a region. All b-labeled transitions enter the region, all c and d labeled transitions exit the region. All e and f labeled transitions do not cross R, i.e., they always connect two states outside the region or two states inside the region.

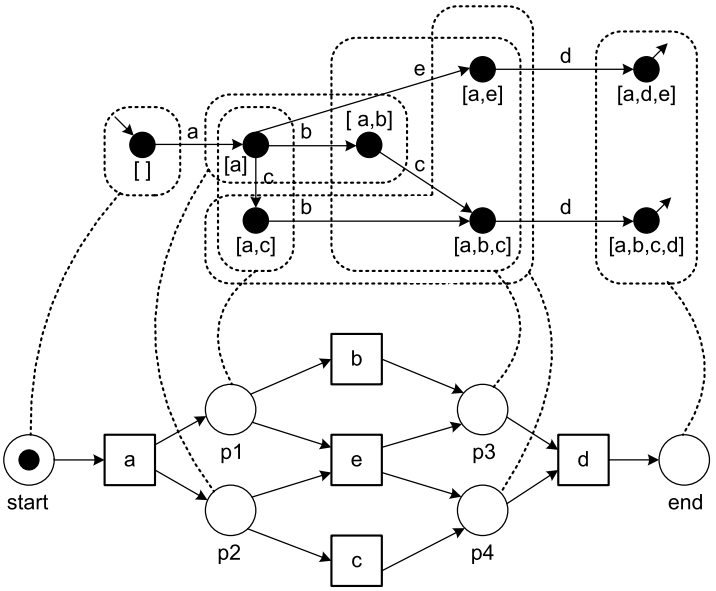

Fig. 7.17 Transition system $TS_{L,l_3^{state}()}$ derived from $L_1 = [\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^2, \langle a, e, d\rangle]$ is converted into a Petri net using state-based regions

By definition, the union of two regions is again a region. Therefore, we are only interested in *minimal* regions. The basic idea is that each minimal region R corresponds to a place p_R in a Petri net as shown in Fig. 7.16. The activities entering the region become Petri-net transitions having p_R as output place, activities leaving the region become output transitions of p_R, and activities that do not cross the region correspond to Petri-net transitions not connected to p_R. Hence, the minimal regions fully encode a Petri net.

Figure 7.17 illustrates the concept of state-based regions using a concrete example. By applying Definition 7.3, we find six minimal regions. Consider for example $R_1 = \{[a], [a, c]\}$. All a labeled transitions in the transition system enter R_1 (there is only one), all b labeled transitions exit R_1 (there are two such transitions), all e labeled transitions exit R_1 (there is only one), and all other transitions in the transition system do not cross R_1. Hence, R_1 is a region corresponding to place $p1$ with input transition a and output transitions b and e. $R_2 = \{[a], [a, b]\}$ is another region: a enters R_2, c and e exit R_2, and all other transitions in the transition system do not cross R_2. R_2 is the region corresponding to place $p2$ in Fig. 7.17. In the Petri net constructed based on the six minimal regions, b and c are concurrent.

Figure 7.17 shows a small process with very little concurrency. Therefore, the transition system and Petri net have similar sizes. However, for larger processes with lots of concurrency the reduction can be spectacular. The transition system modeling 10 parallel activities having $2^{10} = 1024$ states and $10 \times 2^{10-1} = 5120$ transitions, can be reduced into a Petri net with only 20 places and 10 transitions.

The transition system in Fig. 7.17 was obtained from log L_1 using state representation function $l_3^{state}()$. In fact, in this example, the discovered process model using this two-step approach is identical to the model discovered by the α-algorithm. This demonstrates that a two-step approach can be used to convert an event log into a Petri net. Therefore, process discovery using transition system construction and state-based regions is an alternative to the approaches presented thus far.

Figure 7.17 only conveys the basic idea behind regions [51]. The synthesis of Petri nets using state-based regions is actually more involved and can be customized to favor particular process patterns. As shown in [34], any finite transition system can be converted into a bisimilar Petri net, i.e., the behaviors of the transition system and Petri net are equivalent even if we consider the moment of choice (see Sect. 6.3). However, for some Petri nets it may be necessary to perform "label splitting". As a result the Petri net may have multiple transitions referring to the same activity. This way the WF-net shown in Fig. 6.20 can be discovered. Moreover, it is also possible to enforce the resulting Petri net to have particular properties, e.g., free-choice [45]. See [165] for more information about the two-step approach.

Classical state-based regions aim at producing a Petri net that is bisimilar to the transition system. This means that while constructing the Petri net the behavior is not generalized. Therefore, it is important to select a coarser state representation function when constructing the transition system. For larger processes, a state representation function like $l_1^{state}()$ definitely results in an overfitting model that can only replay the log without any form of generalization. Many abstractions (i.e., state representation functions) are possible to balance between overfitting and underfitting. In [165], these are described systematically.

7.4.3 Process Discovery Using Language-Based Regions

As illustrated by Fig. 7.16, the goal of state-based regions is to determine the places in a Petri net. Language-based regions also aim at finding such places but do not use a transition system as input; instead some "language" is used as input. Several techniques and variants of the problem have been defined. In this section we only present the basic idea and refer to literature for details [16, 19, 28, 170].

Suppose, we have an event log in which the events refer to a set of activities A. For this log one could construct a Petri net $N_\emptyset$ with the set of transitions being A and no places. Since a transition without any input places is continuously enabled, this Petri net is able to reproduce the log. In fact, the Petri net $N_\emptyset$ can reproduce any log over A. In Sect. 6.4.3, we referred to such a model as the "flower model". There we added places and transitions to model this behavior in terms of a WF-net. However, the idea is the same. Adding places to the Petri net $N_\emptyset$ can only limit the behavior.

Consider, for example, place p_R in Fig. 7.18. Removing place p_R will not remove any behavior. However, adding p_R may remove behavior possible in the Petri net without this place. The behavior gets restricted when a place is empty while

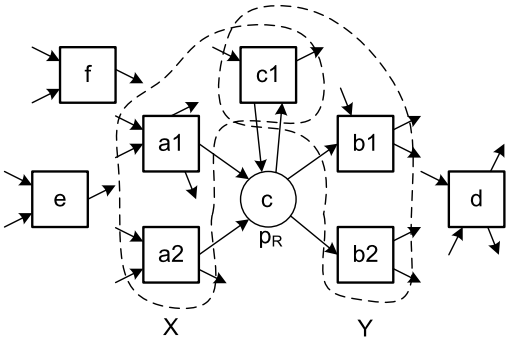

Fig. 7.18 Region $R = (X, Y, c)$ corresponding to place p_R: $X = \{a1, a2, c1\} = \bullet p_R$, $Y = \{b1, b2, c1\} = p_R\bullet$, and c is the initial marking of p_R

one of its output transitions wants to consume a token from it. For example, $b1$ is blocked if p_R is unmarked while all other input places of $b1$ are marked. Suppose now that we have a set of traces L. If these traces are possible in the net with place p_R, then they are also possible in the net without p_R. The reverse does not always hold. This triggers the question whether p_R can be added without disabling any of the traces in L. This is what regions are all about.

Definition 7.4 (Language-based region) Let $L \in \mathbb{B}(\mathscr{A}^*)$ be a simple event log. $R = (X, Y, c)$ is a *region* of L if and only if:

- $X \subseteq \mathscr{A}$ is the set of input transitions of R;
- $Y \subseteq \mathscr{A}$ is the set of output transitions of R;
- $c \in \{0, 1\}$ is the initial marking of R; and
- For any $\sigma \in L$, $k \in \{1, \ldots, |\sigma|\}$, $\sigma_1 = hd^{k-1}(\sigma)$, $a = \sigma(k)$, $\sigma_2 = hd^k(\sigma) = \sigma_1 \oplus a$:

$$c + \sum_{t \in X} \partial_{multiset}(\sigma_1)(t) - \sum_{t \in Y} \partial_{multiset}(\sigma_2)(t) \geq 0$$

$R = (X, Y, c)$ is a region of L if and only if inserting a place p_R with $\bullet p_R = A$, $p_R\bullet = B$, and initially c tokens does not disable the execution of any of the traces in L. To check this, Definition 7.4 inspects all events in the event log. Let $\sigma \in L$ be a trace in the log. $a = \sigma(k)$ is the k-th event in this trace. This event should not be disabled by place p_R. Therefore, we calculate the number of tokens $M(p_R)$ that are in this place just before the occurrence of the k-th event.

$$M(p_R) = c + \sum_{t \in X} \partial_{multiset}(\sigma_1)(t) - \sum_{t \in Y} \partial_{multiset}(\sigma_1)(t)$$

$\sigma_1 = hd^{k-1}(\sigma)$ is the partial trace of events that occurred before the occurrence of the k-th event. $\partial_{multiset}(\sigma_1)$ converts this partial trace into a multi-set. $\partial_{multiset}(\sigma_1)$ is also known as the *Parikh vector* of σ_1. $\sum_{t \in X} \partial_{multiset}(\sigma_1)(t)$ counts the number of tokens produced for place p_R, $\sum_{t \in Y} \partial_{multiset}(\sigma_1)(t)$ counts the number of tokens consumed from this place, and c is the initial number of tokens in p_R. Therefore,

$M(p_R)$ is indeed the number of tokens in p_R just before the occurrence of the k-th event. This number should be positive. In fact, there should be at least one token in p_R if $a \in Y$. In other words, $M(p_R)$ minus the number of tokens consumed from p_R by the k-th event should be non-negative. Hence

$$M(p_R) - \sum_{t \in Y} \partial_{multiset}\big(\langle a \rangle\big)(t) = c + \sum_{t \in X} \partial_{multiset}(\sigma_1)(t) - \sum_{t \in Y} \partial_{multiset}(\sigma_2)(t) \geq 0$$

This shows that a region R, according to Definition 7.4, indeed corresponds to a so-called *feasible place* p_R, i.e., a place that can be added without disabling any of the traces in the event log.

The requirement stated in Definition 7.4 can also be formulated in terms of an inequation system. To illustrate this we use an example log from Chap. 6,

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

There are five activities. For each activity t we introduce two variables, x_t and y_t. $x_t = 1$ if transition t produces a token for p_R and $x_t = 0$ if not. $y_t = 1$ if transition t consumes a token from p_R and $y_t = 0$ if not. A potential region $R = (X, Y, c)$ corresponds to an assignment for all of these variables: $x_t = 1$ if $t \in X$, $x_t = 0$ if $t \notin X$, $y_t = 1$ if $t \in Y$, $y_t = 0$ if $t \notin Y$. The requirement stated in Definition 7.4 can now be reformulated in terms of the variables x_a, x_b, x_c, x_d, x_e, y_a, y_b, y_c, y_d, y_e, and c:

$$\begin{aligned}
c - y_a &\geq 0 \\
c + x_a - (y_a + y_c) &\geq 0 \\
c + x_a + x_c - (y_a + y_c + y_d) &\geq 0 \\
c - y_b &\geq 0 \\
c + x_b - (y_b + y_c) &\geq 0 \\
c + x_b + x_c - (y_b + y_c + y_e) &\geq 0 \\
c, x_a, \ldots, x_e, y_a, \ldots, y_e &\in \{0, 1\}
\end{aligned}$$

Note that these inequations are based on all non-empty prefixes of $\langle a, c, d \rangle$ and $\langle b, c, e \rangle$. Any solution of this linear inequation system corresponds to a region. Some example solutions are:

$$\begin{aligned}
&R_1 = \big(\emptyset, \{a, b\}, 1\big) \\
&\quad c = y_a = y_b = 1, \qquad x_a = x_b = x_c = x_d = x_e = y_c = y_d = y_e = 0 \\
&R_2 = \big(\{a, b\}, \{c\}, 0\big) \\
&\quad x_a = x_b = y_c = 1, \qquad c = x_c = x_d = x_e = y_a = y_b = y_d = y_e = 0 \\
&R_3 = \big(\{c\}, \{d, e\}, 0\big) \\
&\quad x_c = y_d = y_e = 1, \qquad c = x_a = x_b = x_d = x_e = y_a = y_b = y_c = 0
\end{aligned}$$

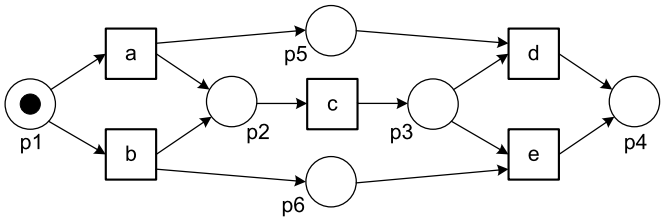

Fig. 7.19 WF-net constructed using regions $R_1, \ldots, R_6$: $p1$ corresponds to $R_1 = (\emptyset, \{a, b\}, 1)$, $p2$ corresponds to $R_2 = (\{a, b\}, \{c\}, 0)$, etc.

$$
\begin{aligned}
&R_4 = \big(\{d, e\}, \emptyset, 0\big)\\
&\quad x_d = x_e = 1, \qquad c = x_a = x_b = x_c = y_a = y_b = y_c = y_d = y_e = 0\\
&R_5 = \big(\{a\}, \{d\}, 0\big)\\
&\quad x_a = y_d = 1, \qquad c = x_b = x_c = x_d = x_e = y_a = y_b = y_c = y_e = 0\\
&R_6 = \big(\{b\}, \{e\}, 0\big)\\
&\quad x_b = y_e = 1, \qquad c = x_a = x_c = x_d = x_e = y_a = y_b = y_c = y_d = 0
\end{aligned}
$$

Consider, for example, $R_6 = (\{b\}, \{e\}, 0)$. This corresponds to the solution $x_b = y_e = 1$ and $c = x_a = x_c = x_d = x_e = y_a = y_b = y_c = y_d = 0$. If we fill out the values in the inequation system, we can see that this is indeed a solution. If we construct a Petri net based on these six regions, we obtain the WF-net shown in Fig. 7.19.

Suppose that the trace $\langle a, c, e\rangle$ is added to event log L_9. This results in three additional inequations:

$$
\begin{aligned}
c - y_a &\geq 0\\
c + x_a - (y_a + y_c) &\geq 0\\
c + x_a + x_c - (y_a + y_c + y_e) &\geq 0
\end{aligned}
$$

Only the last inequation is new. Because of this inequation, $x_b = y_e = 1$ and $c = x_a = x_c = x_d = x_e = y_a = y_b = y_c = y_d = 0$ is no longer a solution. Hence, $R_6 = (\{b\}, \{e\}, 0)$ is not a region anymore and place $p6$ needs to be removed from the WF-net shown in Fig. 7.19. After removing this place, the resulting WF-net indeed allows for $\langle a, c, e\rangle$.

One of the problems of directly applying language-based regions is that the linear inequation system has many solutions. Few of these solutions correspond to sensible places. For example, $x_a = x_b = y_d = y_e = 1$ and $c = x_c = x_d = x_e = y_a = y_b = y_c = 0$ also defines a region: $R_7 = (\{a, b\}, \{d, e\}, 0)$. However, adding this place to Fig. 7.19 would only clutter the diagram. Another example is $c = x_a = x_b = y_c = 1$ and $x_c = x_d = x_e = y_a = y_b = y_d = y_e = 0$, i.e., region $R_8 = (\{a, b\}, \{c\}, 1)$. This region is a weaker variant R_2 as the place is initially marked.

Another problem is that classical techniques for language-based regions aim at a Petri net that does not allow for any behavior not seen in the log [19]. This means that the log is considered to be complete. As shown before, this is very unrealistic and results in models that are complex and overfitting. To address these problems dedicated techniques have been proposed. For instance, in [170] it is shown how to avoid overfitting and how to ensure that the resulting model has desirable properties (WF-net, free-choice, etc.). Nevertheless, pure region-based techniques tend to have problems handling noise and incompleteness. Therefore, combinations of heuristic mining and region-based techniques seem more suitable for practical applications.

7.5 Inductive Mining

A range of *inductive process discovery* techniques exist for the *process trees* introduced in Sect. 3.2.8 [88–91]. Whereas Petri nets, WF-nets, BPMN models, EPCs, YAWL models, and UML activity diagrams may suffer from deadlocks, livelocks, and other anomalies, process trees are *sound by construction*. This section introduces the basic inductive mining approach. The inductive mining framework is highly extendible and allows for many variants of the basic approach. The "family" of inductive mining techniques includes members that can handle infrequent behavior and deal with huge models and logs while ensuring formal correctness criteria such as the ability to rediscover the original model (in the limit). The results returned by these techniques can easily be converted to other notations ranging from Petri nets and BPMN models to process calculi and statecharts. Inductive mining is currently one of the leading process discovery approaches due to its flexibility, formal guarantees and scalability.

7.5.1 Inductive Miner Based on Event Log Splitting

Given a simple event log $L \in \mathbb{B}(A^*)$ (i.e., a multi-set of traces over some set of activities A) we would like to discover a process tree $Q \in \mathscr{Q}_A$. Consider, for example, event log $L_1 = [\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^2, \langle a, e, d\rangle]$ consisting of 6 cases and 23 events. The α-algorithm creates the WF-net $N_1 = \alpha(L_1)$ shown in Fig. 7.20 (left-hand side). The basic *Inductive Miner* (IM) will produce the equivalent process tree $Q_1 = \text{IM}(L_1) = \rightarrow(a, \times(\wedge(b, c), e), d)$ also shown in Fig. 7.20 (right-hand side). The process tree can be automatically converted into the WF-net produced by the α-algorithm using the conversion shown in Fig. 3.18 followed by a reduction removing superfluous silent transitions. Any process tree can be converted to an equivalent WF-net, BPMN model, etc. Moreover, the basic Inductive Miner (IM) can discover a much wider class of processes and learn "correct" process models in situations where the α-algorithm and many other algorithms fail.

We use several simple examples to explain the approach. For clarity we first assume that there are no duplicate or silent activities, i.e., in the process trees used

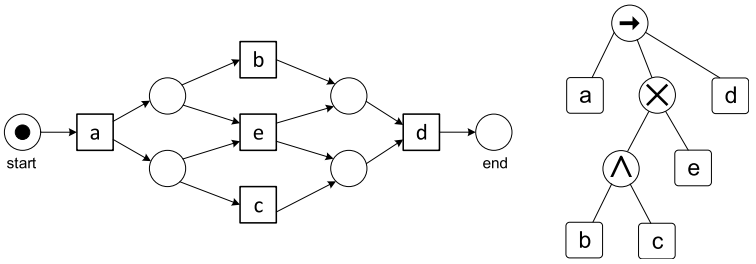

Fig. 7.20 WF-net N_1 (left) and process tree Q_1 (right) discovered for $L_1 = [\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^2, \langle a, e, d\rangle]$

to generate the example logs there are no two leaves with the same activity label and no leaves with a τ label. Later we relax this assumption.

The basic IM algorithm uses the *directly-follows graph* that corresponds to the "directly follows" relation ($>_L$) used by the α-algorithm. Other elements such as "eventually follows" or the dependency measures used for the dependency graph can also be used in the Inductive Miner (IM) framework. Frequencies provide important information for process discovery. However, we start by looking at ingredients similar to the ones used by the α-algorithm: the directly follows relation, start activities, and end activities.

Definition 7.5 (Directly-follows graph) Let L be an event log, i.e., $L \in \mathbb{B}(\mathscr{A}^*)$. The *directly-follows graph* of L is $G(L) = (A_L, \mapsto_L, A_L^{start}, A_L^{end})$ with:

- $A_L = \{a \in \sigma \mid \sigma \in L\}$ is the set of activities in L,
- $\mapsto_L = \{(a, b) \in A \times A \mid a >_L b\}$ is the directly follows relation,[3]
- $A_L^{start} = \{a \in A \mid \exists_{\sigma \in L} a = first(\sigma)\}$ is the set of start activities, and
- $A_L^{end} = \{a \in A \mid \exists_{\sigma \in L} a = last(\sigma)\}$ is the set of end activities.

The IM algorithm iteratively splits the initial event log into smaller *sublogs*. For any sublog L we can create a directly-follows graph $G(L)$. $a \mapsto_L b$ if a was directly followed by b somewhere in L. $a \not\mapsto_L b$ if a was never directly followed by b. $\mapsto_L^+$ is the transitive closure of $\mapsto_L$. $a \mapsto_L^+ b$ if there is a non-empty *path* from a to b in $G(L)$, i.e., there exists a sequence of activities $a_1, a_2, \ldots, a_k$ such that $k \geq 2$, $a_1 = a$ and $a_k = b$ and $a_i \mapsto_L a_{i+1}$ for $i \in \{1, \ldots, k-1\}$. $a \not\mapsto_L^+ b$ if there is no path from a to b in the directly-follows graph.

To understand how the IM algorithm learns $Q_1 = \text{IM}(L_1) = \rightarrow(a, \times(\wedge(b, c), e), d)$ shown in Fig. 7.20 from event log $L_1 = [\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^2, \langle a, e, d\rangle]$, consider Fig. 7.21 and Fig. 7.22. $G_1 = G(L_1)$ in Fig. 7.21 is the directly-follows graph of L_1. Note that $a \mapsto_{L_1} b$ because of the arc between a and b in G_1. $a \not\mapsto_{L_1} d$ and $a \mapsto_{L_1}^+ d$ because there is a path from a to d, but no arc between a and d.

[3] $a >_L b$ 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 $t_{i+1} = b$ (see Definition 6.3).

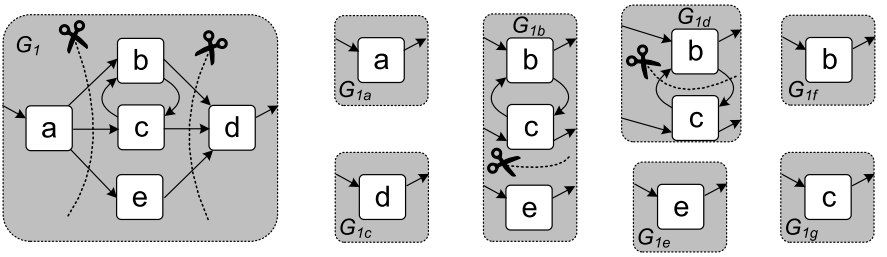

Fig. 7.21 G_1 is the directly-follows graph for $L_1 = [\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^2, \langle a, e, d\rangle]$. The event log is recursively cut into smaller sublogs using the directly-follows graphs of these sublogs

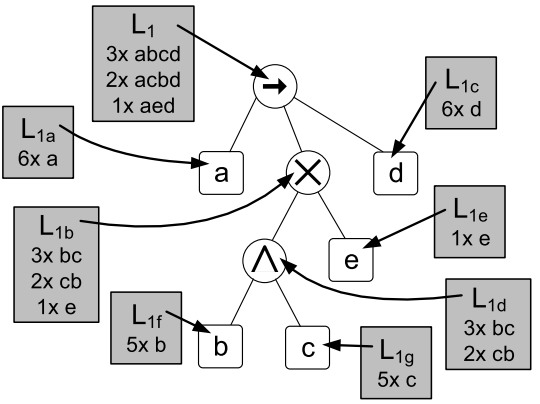

Fig. 7.22 The different sublogs created when learning process tree $Q_1 = \rightarrow(a, \times(\wedge(b, c), e), d)$ for $L_1 = [\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^2, \langle a, e, d\rangle]$

$A_{L_1}^{start} = \{a\}$ as denoted by the incoming arc. $A_{L_1}^{end} = \{d\}$ as denoted by the outgoing arc.

We would like to split L_1 recursively until we find sublogs of the form $[\langle x\rangle^k]$, i.e., sublogs corresponding to the execution of activity x. To find out how to split the event log in each step, we try to find so-called *cuts* in the directly-follows graph of the (sub)log we would like to split. We consider *exclusive-choice cuts*, *sequence cuts*, *parallel cuts*, and *redo-loop cuts* corresponding to the four process tree operators ($\times$, $\rightarrow$, $\wedge$, and $\circlearrowleft$).

Directly-follows graph $G_1 = G(L_1)$ in Fig. 7.21 is cut into three smaller directly-follows graphs (G_{1a}, G_{1b}, and G_{1c}) using *sequence cut* $(\rightarrow, \{a\}, \{b, c, e\}, \{d\})$. The cut splits the set of activities in three disjoint subsets such that arcs only go from left to right. The formal definition of a sequence cut is given later. Based on the sequence cut, event log L_1 is split into $L_{1a} = [\langle a\rangle^6]$, $L_{1b} = [\langle b, c\rangle^3, \langle c, b\rangle^2, \langle e\rangle]$, and $L_{1c} = [\langle d\rangle^6]$. These sublogs are created by projecting the original event log on the three disjoint subsets of activities in cut $(\rightarrow, \{a\}, \{b, c, e\}, \{d\})$. Note that each event in L_1 ends up in precisely one of the sublogs. Using the three sublogs, we create three new directly-follows graphs, $G_{1a} = G(L_{1a})$, $G_{1b} = G(L_{1b})$, and $G_{1c} = G(L_{1c})$. These graphs are shown in Fig. 7.21. $G_{1a} = G(L_{1a})$ and $G_{1c} = G(L_{1c})$ represent

base cases, i.e., subprocesses consisting of a single activity executed once per case. G_{1b} is not a base case and needs to be split further.

$\text{IM}(L_1) = \rightarrow(\text{IM}(L_{1a}), \text{IM}(L_{1b}), \text{IM}(L_{1c}))$ because of the sequence cut. Since $\text{IM}(L_{1a}) = a$ and $\text{IM}(L_{1c}) = d$, this can be rewritten as $\text{IM}(L_1) = \rightarrow(a, \text{IM}(L_{1b}), d)$. Next we compute $\text{IM}(L_{1b})$ using $G_{1b} = G(L_{1b})$ in Fig. 7.21. Directly-follows graph $G_{1b} = G(L_{1b})$ is cut into two smaller directly-follows graphs using *exclusive-choice cut* $(\times, \{b, c\}, \{e\})$. The exclusive-choice cut splits the set of activities in two disjoint subsets such that there are no arcs going from one set to the other (and vice versa). Based on the sequence cut, event log L_{1b} is split into $L_{1d} = [\langle b, c\rangle^3, \langle c, b\rangle^2]$ and $L_{1e} = [\langle e\rangle]$. Again each event ends up in precisely one of the sublogs. However, because of the nature of the exclusive-choice cut, we partitioned the traces based on the activities they contain rather than projecting trace on disjoint activity sets.

$\text{IM}(L_{1b}) = \times(\text{IM}(L_{1d}), \text{IM}(L_{1e}))$ because of the exclusive-choice cut. $\text{IM}(L_{1e}) = e$ corresponds again to the base case: In each trace in the sublog, activity d is executed once (compare G_{1e} with G_{1a} and G_{1c}). It remains to compute $\text{IM}(L_{1d})$. Figure 7.21 shows directly-follows graph $G_{1d} = G(L_{1d})$ which is cut into two smaller directly-follows graphs using *parallel cut* $(\wedge, \{b\}, \{c\})$. The parallel cut splits the set of activities in two disjoint subsets such that every activity in one set is connected to all activities in the other set (and vice versa). Based on the parallel cut, event log L_{1d} is split into $L_{1f} = [\langle b\rangle^5]$ and $L_{1g} = [\langle c\rangle^5]$. $G_{1f} = G(L_{1f})$ and $G_{1g} = G(L_{1g})$ are shown in Fig. 7.21 and correspond to the base case. Hence, $\text{IM}(L_{1f}) = b$ and $\text{IM}(L_{1g}) = c$. Therefore, $\text{IM}(L_{1d}) = \wedge(b, c)$, $\text{IM}(L_{1b}) = \times(\wedge(b, c), e)$, $\text{IM}(L_1) = \rightarrow(a, \times(\wedge(b, c), e), d)$.

Process tree $Q_1 = \text{IM}(L_1) = \rightarrow(a, \times(\wedge(b, c), e), d)$ was computed by recursively applying a sequence cut, an exclusive-choice cut, and a parallel cut. Figure 7.22 shows the process tree and the sublogs created during discovery. The leaves correspond to base cases. The inner nodes correspond to operators used to cut the event log in sublogs.

Definition 7.6 (Cut) Let L be an event log with corresponding directly-follows graph $G(L) = (A_L, \mapsto_L, A_L^{start}, A_L^{end})$. Let $n \geq 1$. An n-ary *cut* of $G(L)$ is a partition of A_L into pairwise disjoint sets $A_1, A_2, \ldots, A_n$: $A_L = \bigcup_{i \in \{1,\ldots,n\}} A_i$ and $A_i \cap A_j = \emptyset$ for $i \neq j$. Notation is $(\oplus, A_1, A_2, \ldots, A_n)$ with $\oplus \in \{\rightarrow, \times, \wedge, \circlearrowleft\}$. For each type of operator ($\rightarrow$, $\times$, $\wedge$, and $\circlearrowleft$) specific conditions apply:

- An *exclusive-choice cut* of $G(L)$ is a cut $(\times, A_1, A_2, \ldots, A_n)$ such that
  - $\forall_{i,j \in \{1,\ldots,n\}} \forall_{a \in A_i} \forall_{b \in A_j}\ i \neq j \Rightarrow a \not\mapsto_L b$.
- A *sequence cut* of $G(L)$ is a cut $(\rightarrow, A_1, A_2, \ldots, A_n)$ such that
  - $\forall_{i,j \in \{1,\ldots,n\}} \forall_{a \in A_i} \forall_{b \in A_j}\ i < j \Rightarrow (a \mapsto_L^+ b \wedge b \not\mapsto_L^+ a)$.
- A *parallel cut* of $G(L)$ is a cut $(\wedge, A_1, A_2, \ldots, A_n)$ such that
  - $\forall_{i \in \{1,\ldots,n\}}\ A_i \cap A_L^{start} \neq \emptyset \wedge A_i \cap A_L^{end} \neq \emptyset$ and
  - $\forall_{i,j \in \{1,\ldots,n\}} \forall_{a \in A_i} \forall_{b \in A_j}\ i \neq j \Rightarrow a \mapsto_L b$.

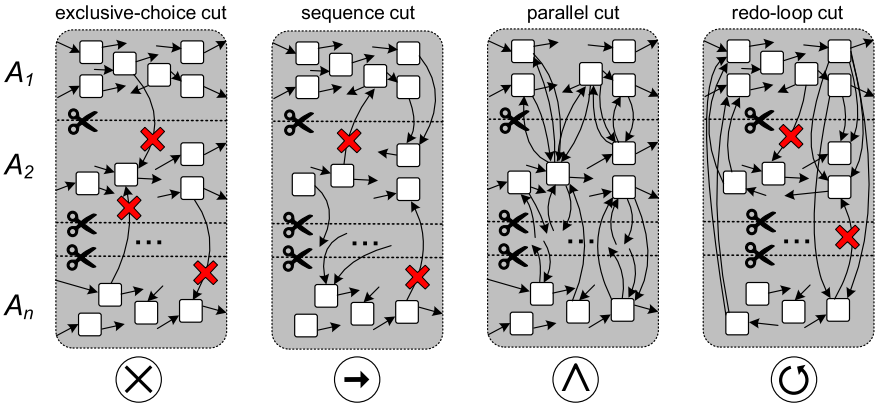

Fig. 7.23 Four types of cuts, $(\oplus, A_1, A_2, \ldots, A_n)$ with $\oplus \in \{\times, \rightarrow, \wedge, \circlearrowleft\}$

- A *redo-loop cut* of $G(L)$ is a cut $(\circlearrowleft, A_1, A_2, \ldots, A_n)$ such that
  - $n \geq 2$,
  - $A_L^{start} \cup A_L^{end} \subseteq A_1$,
  - $\{a \in A_1 \mid \exists_{i \in \{2,\ldots,n\}} \exists_{b \in A_i}\ a \mapsto_L b\} \subseteq A_L^{end}$,
  - $\{a \in A_1 \mid \exists_{i \in \{2,\ldots,n\}} \exists_{b \in A_i}\ b \mapsto_L a\} \subseteq A_L^{start}$,
  - $\forall_{i,j \in \{2,\ldots,n\}} \forall_{a \in A_i} \forall_{b \in A_j}\ i \neq j \Rightarrow a \not\mapsto_L b$,
  - $\forall_{i \in \{2,\ldots,n\}} \forall_{b \in A_i} \exists_{a \in A_L^{end}}\ a \mapsto_L b \Rightarrow \forall_{a' \in A_L^{end}}\ a' \mapsto_L b$, and
  - $\forall_{i \in \{2,\ldots,n\}} \forall_{b \in A_i} \exists_{a \in A_L^{start}}\ b \mapsto_L a \Rightarrow \forall_{a' \in A_L^{start}}\ b \mapsto_L a'$.

A cut $(\oplus, A_1, A_2, \ldots, A_n)$ with $\oplus \in \{\rightarrow, \times, \wedge, \circlearrowleft\}$ of directly-follows graph $G(L)$ is *maximal* if there is no cut $(\oplus, A'_1, A'_2, \ldots, A'_m)$ with $m > n$. Cut $(\oplus, A_1, A_2, \ldots, A_n)$ is called *trivial* if $n = 1$.

The four types of cuts are illustrated in Fig. 7.23. These are based on the characteristics of the four process tree operators assuming that there are no duplicate or silent activities. Definition 3.14 describes the semantics of operator $\oplus \in \{\times, \rightarrow, \wedge, \circlearrowleft\}$. It is easy to verify that these operators indeed leave the "fingerprints" shown in Fig. 7.23.

Consider four simple process trees, $Q_{ab} = \times(a, b)$, $Q_{cd} = \rightarrow(c, d)$, $Q_{ef} = \wedge(e, f)$, and $Q_{gh} = \circlearrowleft(g, h)$. $\mathscr{L}(Q_{ab}) = \{\langle a\rangle, \langle b\rangle\}$, $\mathscr{L}(Q_{cd}) = \{\langle c, d\rangle\}$, $\mathscr{L}(Q_{ef}) = \{\langle e, f\rangle, \langle f, e\rangle\}$, $\mathscr{L}(Q_{gh}) = \{\langle g\rangle, \langle g, h, g\rangle, \langle g, h, g, h, g\rangle, \ldots\}$.

Consider now the directly-follows graph of an event log L generated by $\times(Q_{ab}, Q_{cd}, Q_{ef}, Q_{gh})$ that is *directly-follows complete*. An event log L generated from a process tree is directly-follows complete if directly-follows graph $G(L)$ is maximal, i.e., all activities, all start activities, all end activities, and all possible direct successions have been observed. Clearly, the exclusive-choice cut $(\times, \{a, b\}, \{c, d\}, \{e, f\}, \{g, h\})$ meets the requirement stated in Definition 7.6 for any directly-follows complete log. For example, $a \not\mapsto_L c$, $d \not\mapsto_L h$, etc. The activities in the pairwise disjoint activity sets never follow one another directly.

Next, we consider the directly-follows graph of an event log L generated by $\rightarrow(Q_{ab}, Q_{cd}, Q_{ef}, Q_{gh})$ that is directly-follows complete. The sequence cut $(\rightarrow, \{a, b\}, \{c, d\}, \{e, f\}, \{g, h\})$ meets the requirements stated in Definition 7.6. For example, $a \mapsto_L^+ c$, $c \not\mapsto_L^+ a$, $a \mapsto_L^+ e$, $e \not\mapsto_L^+ a$, etc. Activities in different subsets need to be strictly ordered to apply this cut.

The directly-follows graph of a directly-follows complete event log L generated by $\wedge(Q_{ab}, Q_{cd}, Q_{ef}, Q_{gh})$ allows for parallel cut $(\wedge, \{a, b\}, \{c, d\}, \{e, f\}, \{g, h\})$. The first requirement stated in Definition 7.6 ensures that each of the activity subsets has at least one start and one end activity. The second requirement states that any two activities in different subsets can directly follow one another (e.g., $a \mapsto_L c$, $c \mapsto_L a$, $a \mapsto_L e$, $e \mapsto_L a$, etc.). Both requirements are satisfied by the nature of parallel composition.

The directly-follows graph of a directly-follows complete event log L generated by $\circlearrowleft(Q_{ab}, Q_{cd}, Q_{ef}, Q_{gh})$ allows for redo-loop cut $(\circlearrowleft, \{a, b\}, \{c, d\}, \{e, f\}, \{g, h\})$. Each of the seven requirements in Definition 7.6 is satisfied. All start and end activities are in the "do part" of the redo-loop, i.e., $A_L^{start} \cup A_L^{end} = \{a, b\} \cup \{a, b\} \subseteq \{a, b\}$. All connections run via a and b, e.g., $a \mapsto_L c$, $d \mapsto_L a$, $b \mapsto_L c$, $d \mapsto_L b$, etc. In a redo loop, the directly-follows graph must contain a clear set of start and end activities. All connections between the different child nodes must go through these activities.

The IM algorithm works as follows. Given an event log, the directly-follows graph is constructed. If there is a non-trivial exclusive-choice cut, then a maximal exclusive-choice cut is applied splitting the event log into smaller event logs. If there is no non-trivial exclusive-choice cut, but there is a non-trivial sequence cut, then a maximal sequence cut is applied splitting the event log into smaller event logs. If there are no non-trivial exclusive-choice and sequence cuts, but there is a non-trivial parallel cut, then a maximal parallel cut is applied splitting the event log into smaller event logs. If there are no non-trivial exclusive-choice, sequence and parallel cuts, but there is a redo-loop cut, then a maximal redo-loop cut is applied splitting the event log into smaller event logs. After splitting the event log into sublogs the procedure is repeated until a base case (sublog with only one activity) is reached.

How the event log is split into sublogs depends on the operator. In case of an exclusive-choice cut, the traces are split as a whole. In case of a sequence cut and parallel cut, the traces are projected on the respective sets of activities, i.e., each sublog has a projected trace for each trace in the log that needs to be split. In case of a redo-loop cut, the loops are unfolded and a trace is created for every iteration. Empty traces are handled in a dedicated manner (based on the operator) and result in the insertion of τ activities. For example, exclusive choice cut $(\times, A_1, A_2, \ldots, A_n)$ may result in $\times(Q_1, Q_2, \ldots, Q_n, \tau)$ if there are empty traces in the log to be split.

If there are no non-trivial cuts meeting the requirements in Definition 7.6, a *fall-through* is selected. The part that cannot be split is presented by a so-called *flower model* ("anything can happen"), similar to the one introduced in Sect. 6.4.3. For example, the flower model in Fig. 6.23 can be represented as process tree $\circlearrowleft(\tau, a, b, c, d, e, f, g, h)$ allowing for any trace involving activities a–h.

Suppose the sublog L' for which no cut is applicable contains activities $\{a_1, a_2, \ldots, a_m\}$. Fall-through $\text{IM}(L') = \circlearrowleft(\tau, a_1, a_2, \ldots, a_m)$ is selected, i.e., the subpro-

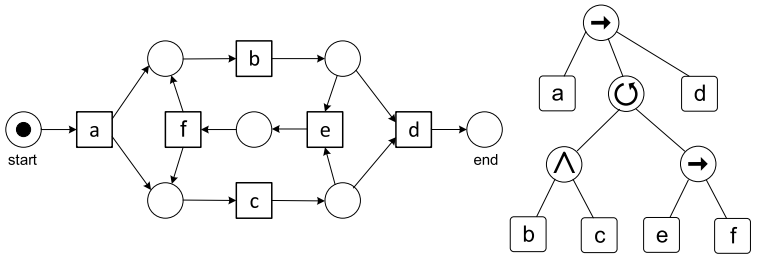

Fig. 7.24 WF-net N_2 (*left*) and process tree Q_2 (*right*) discovered for $L_2 = [\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^4, \langle a, b, c, e, f, b, c, d\rangle^2, \langle a, c, b, e, f, b, c, d\rangle^2, \langle a, b, c, e, f, c, b, d\rangle, \langle a, c, b, e, f, b, c, e, f, c, b, d\rangle]$

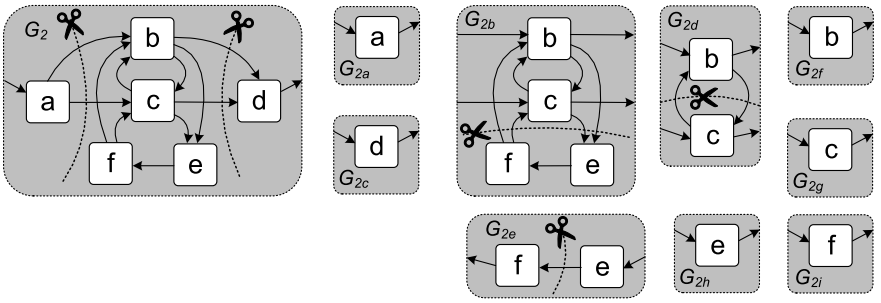

Fig. 7.25 G_2 is the directly-follows graph for L_2. The other directly-follows graphs correspond to the various sublogs

cess is represented by a subtree that allows for any behavior involving activities $\{a_1, a_2, \ldots, a_m\}$. The fall-through serves as a last resort ensuring fitness, but possibly resulting in lower precision.

In the base case, the sublog L' contains only events corresponding to a particular activity, say a. If the sublog is of the form $L' = [\langle a\rangle^k]$ with $k \geq 1$ (i.e., a occurs once in each trace), then $\text{IM}(L') = a$. If the sublog is of the form $L' = [\langle\,\rangle^k, \langle a\rangle^l]$ with $k, l \geq 1$, then $\text{IM}(L') = \times(a, \tau)$ because a is sometimes skipped. If a is executed at least once in each trace in the sublog and sometimes multiple times (e.g., $L' = [\langle a\rangle^9, \langle a, a\rangle^2, \langle a, a, a\rangle]$), then $\text{IM}(L') = \circlearrowleft(a, \tau)$. In all other cases (e.g., $L' = [\langle\,\rangle^3, \langle a\rangle^4, \langle a, a, a\rangle]$), $\text{IM}(L') = \circlearrowleft(\tau, a)$ because a is executed zero or more times in the traces of sublog L.

To illustrate the IM algorithm better, we consider a slightly larger event log,

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

This log could have been generated by the WF-net N_2 or process tree $Q_2 = \rightarrow(a, \circlearrowleft(\wedge(b, c), \rightarrow(e, f)), d)$ both shown in Fig 7.24.

The IM algorithm starts by creating the directly-follows graph $G_2 = G(L_2)$ for event log L_2 (see Fig. 7.25). Since there is no non-trivial exclusive-choice cut, we

try a sequence cut. The maximal sequence cut $(\rightarrow, \{a\}, \{b, c, e, f\}, \{d\})$ is shown in Fig. 7.25. Based on this cut, three sublogs are created:

$$L_{2a} = \left[\langle a\rangle^{13}\right]$$

$$L_{2b} = \left[\langle b, c\rangle^{3}, \langle c, b\rangle^{4}, \langle b, c, e, f, b, c\rangle^{2}, \langle c, b, e, f, b, c\rangle^{2}, \langle b, c, e, f, c, b\rangle, \langle c, b, e, f, b, c, e, f, c, b\rangle\right]$$

$$L_{2c} = \left[\langle d\rangle^{13}\right]$$

Event log L_{2a} (L_{2c}) has directly-follows graph $G_{2a} = G(L_{2a})$ ($G_{2c} = G(L_{2c})$). Both correspond to base cases and are represented by subtrees $\text{IM}(L_{2a}) = a$ and $\text{IM}(L_{2c}) = d$. $G_{2b} = G(L_{2b})$ in Fig. 7.25 is the directly-follows graph for sublog L_{2b}. There are no non-trivial exclusive-choice, sequence or parallel cuts. Therefore, we apply the maximal redo-loop cut $(\circlearrowleft, \{b, c\}, \{e, f\})$ shown in Fig. 7.25. Note that all seven requirements stated in Definition 7.6 are satisfied. Using the redo-loop cut, sublog L_{2b} is split into two smaller sublogs, $L_{2d} = [\langle b, c\rangle^{11}, \langle c, b\rangle^{9}]$ and $L_{2e} = [\langle e, f\rangle^{7}]$. Note that some traces in L_{2b} correspond to multiple traces in L_{2d} and L_{2e}. Consider, for example, $\langle c, b, e, f, b, c, e, f, c, b\rangle \in L_{2b}$ which is split into five smaller traces ($\langle c, b\rangle$, $\langle e, f\rangle$, $\langle b, c\rangle$, $\langle e, f\rangle$, and $\langle c, b\rangle$) distributed over L_{2d} and L_{2e}. Subsequently, the IM algorithm selects the parallel cut $(\wedge, \{b\}, \{c\})$ in G_{2d}, the directly-follows graph created for L_{2d}. The resulting sublogs $L_{2f} = [\langle b\rangle^{20}]$ and $L_{2g} = [\langle c\rangle^{20}]$ correspond to base cases. Hence, $\text{IM}(L_{2f}) = b$, $\text{IM}(L_{2g}) = c$, and $\text{IM}(L_{2d}) = \wedge(b, c)$. G_{2e} is the directly-follows graph created for L_{2e}. The IM algorithm selects the sequence cut $(\rightarrow, \{e\}, \{f\})$. The resulting sublogs $L_{2h} = [\langle e\rangle^{7}]$ and $L_{2i} = [\langle f\rangle^{7}]$ correspond to base cases. Hence, $\text{IM}(L_{2h}) = e$, $\text{IM}(L_{2i}) = f$, and $\text{IM}(L_{2e}) = \rightarrow(e, f)$. $\text{IM}(L_{2b}) = \circlearrowleft(\wedge(b, c), \rightarrow(e, f))$. By combining the results for the subtrees, we find $Q_2 = \text{IM}(L_2) = \rightarrow(a, \circlearrowleft(\wedge(b, c), \rightarrow(e, f)), d)$.

Finally, we revisit the example from Chap. 2 (cf. Table 2.2). The IM algorithm is able to discover the same model as the α-algorithm. For any directly-follows complete event log generated by the WF-net in Fig. 7.27, the IM algorithm discovers a process tree equivalent to $\rightarrow(a, \circlearrowleft(\rightarrow(\wedge(\times(b, c), d), e), f), \times(g, h))$.

7.5.2 Characteristics of the Inductive Miner

The IM algorithm returns a specific process tree. However, there may be several process trees having a same behavior. For example, $\times(a, b, c)$ and $\times(c, \times(b, a))$ are indistinguishable, i.e., $\mathscr{L}(\times(a, b, c)) = \mathscr{L}(\times(c, \times(b, a))) = \{\langle a\rangle, \langle b\rangle, \langle c\rangle\}$. $\wedge(a, b)$ and $\wedge(b, a, \tau)$ are also indistinguishable, i.e., $\mathscr{L}(\wedge(a, b)) = \mathscr{L}(\wedge(b, a, \tau)) = \{\langle a, b\rangle, \langle b, a\rangle\}$. We consider two process trees Q_1 and Q_2 to be *equivalent* if $\mathscr{L}(Q_1) = \mathscr{L}(Q_2)$ (i.e., trace equivalence).

The ordering of the children of a parallel or exclusive choice operator does not matter. We assume the IM algorithm to be deterministic and pick a particular order. Therefore, we can write $\text{IM}(L)$.

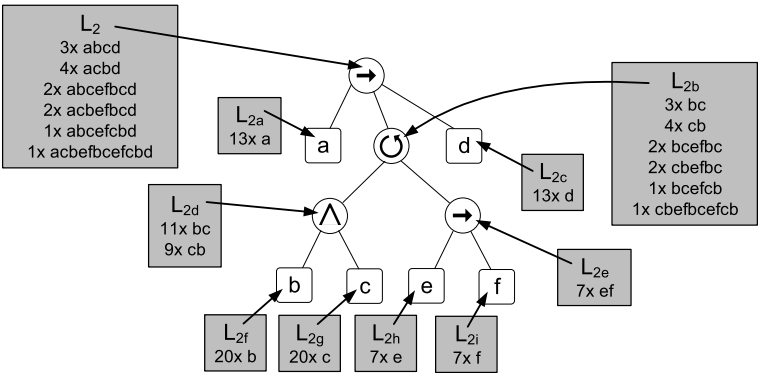

Fig. 7.26 The different sublogs created when learning process tree $Q_2 = \rightarrow(a, \circlearrowleft(\wedge(b, c), \rightarrow(e, f)), d)$ for $L_2 = [\langle a, b, c, d\rangle^3, \langle a, c, b, d\rangle^4, \langle a, b, c, e, f, b, c, d\rangle^2, \langle a, c, b, e, f, b, c, d\rangle^2, \langle a, b, c, e, f, c, b, d\rangle, \langle a, c, b, e, f, b, c, e, f, c, b, d\rangle]$

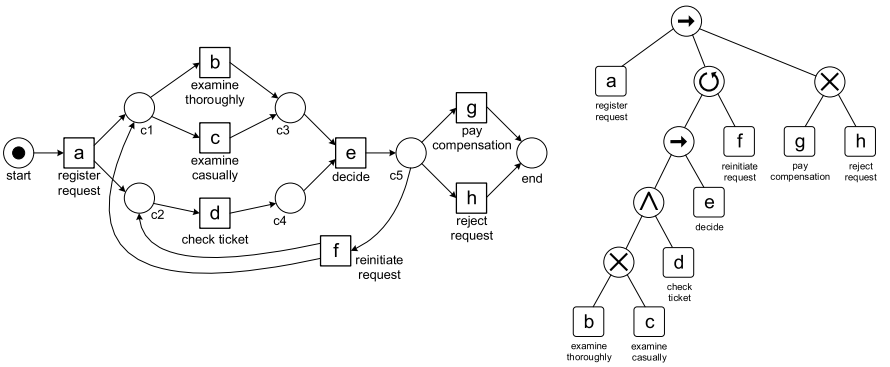

Fig. 7.27 The process model discovered by the α-algorithm for event log $[\langle a, b, d, e, h\rangle, \langle a, d, c, e, g\rangle, \langle a, c, d, e, f, b, d, e, g\rangle, \langle a, d, b, e, h\rangle, \langle a, c, d, e, f, d, c, e, f, c, d, e, h\rangle, \langle a, c, d, e, g\rangle]$ and the corresponding process tree $\rightarrow(a, \circlearrowleft(\rightarrow(\wedge(\times(b, c), d), e), f), \times(g, h))$

A process tree Q is called *language-rediscoverable* by the IM algorithm if for any directly-follows complete event log L generated from Q, $\mathscr{L}(\text{IM}(L)) = \mathscr{L}(Q)$. Recall that an event log L is *directly-follows complete* for process tree Q if the directly-follows graph $G(L)$ is *maximal*, i.e., all activities in Q appear in the log, for every start (end) activity in Q there is a trace starting (ending) with it in the log, and $a \mapsto_L b$ if b can directly follow a in Q.

In [88], it is shown that almost all process trees *without duplicate and silent activities* are *language-rediscoverable* using the basic algorithm described in Sect. 7.5.1. The only exception is the situation where both a redo-loop cut and parallel cut are possible. An example is shown in Fig. 7.28. $Q_{rd} = \circlearrowleft(\wedge(a, b), \wedge(c, d))$ and $Q_{par} = \wedge(\circlearrowleft(a, c), \circlearrowleft(b, d))$ are not trace equivalent, but have the same directly-follows graph for any directly-follows complete event log. Since the IM algorithm

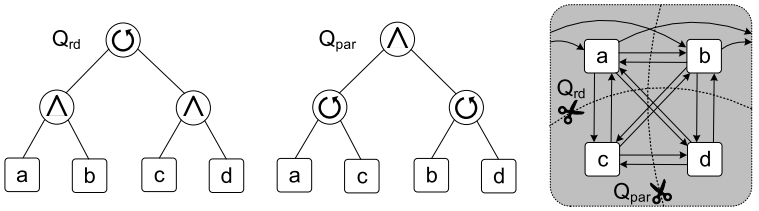

Fig. 7.28 Two process trees $Q_{rd} = \circlearrowleft(\wedge(a,b), \wedge(c,d))$ and $Q_{par} = \wedge(\circlearrowleft(a,c), \circlearrowleft(b,d))$ having different behaviors but identical directly-follows graphs: redo-loop cut $(\circlearrowleft, \{a,b\}, \{c,d\})$ and parallel cut $(\wedge, \{a,c\}, \{b,d\})$ are both possible

only uses the directly-follows graph, it cannot distinguish both processes. Fortunately, this is a rather peculiar situation and still does not jeopardize fitness. In almost all cases, the directly-follows graph is informative enough. For example, if the start and end activities in the "do part" of the loop are disjoint, then language-rediscoverability is guaranteed [88].

Note that $\mathscr{L}(Q_{rd}) \subset \mathscr{L}(Q_{par})$. The IM algorithm selects the parallel cut $(\wedge, \{a,c\}, \{b,d\})$ and returns the more general Q_{par} that also allows for traces like $\langle a,b,c,a\rangle$ and $\langle a,c,a,b,d,b\rangle$ not possible in Q_{rd}.

Importantly, the IM algorithm *always produces a sound process model able to replay the whole event log*. Unlike many other algorithms, fitness is guaranteed. Since the models are block-structured and activities are not duplicated, the models tend to be simple and general (overfitting models are often the result of excessive label splitting). However, the fall-through in the IM algorithm may create underfitting models. This may occur in situations where there is no process tree without duplicate and silent activities generating the observed behavior.

Apart from the very special situation sketched in Fig. 7.28, any process tree without duplicate and silent activities is language-rediscoverable. When allowing for duplicate and silent activities such guarantees are more difficult to provide. The basic IM algorithm described in Sect. 7.5.1 never duplicates activities. Silent activities are only introduced for base cases and empty traces, e.g., $\times(\tau, a)$ if a can be skipped $\circlearrowleft(a,\tau)$ if a can be repeated, and $\circlearrowleft(\tau,a)$ if a can be skipped and repeated. In the presence of duplicate and silent activities, the directly-follows graph provides insufficient information to ensure language-rediscoverability. However, weaker guarantees such as the ability to replay the event log without any problems still hold.

To get an understanding of the limitations of the basic IM algorithm, we consider the example logs used to introduce the α-algorithm in Chap. 6. As already shown, the process trees discovered by the IM algorithm for logs L_1 and L_2 are behaviorally equivalent to the WF-nets discovered by the α-algorithm (see Fig 7.20 and Fig 7.24). Some other logs used in Chap. 6 are considered next:

$$L_3 = \big[\langle a,b,c,d,e,f,b,d,c,e,g\rangle, \langle a,b,d,c,e,g\rangle^2,$$
$$\langle a,b,c,d,e,f,b,c,d,e,f,b,d,c,e,g\rangle\big]$$
$$L_4 = \big[\langle a,c,d\rangle^{45}, \langle b,c,d\rangle^{42}, \langle a,c,e\rangle^{38}, \langle b,c,e\rangle^{22}\big]$$

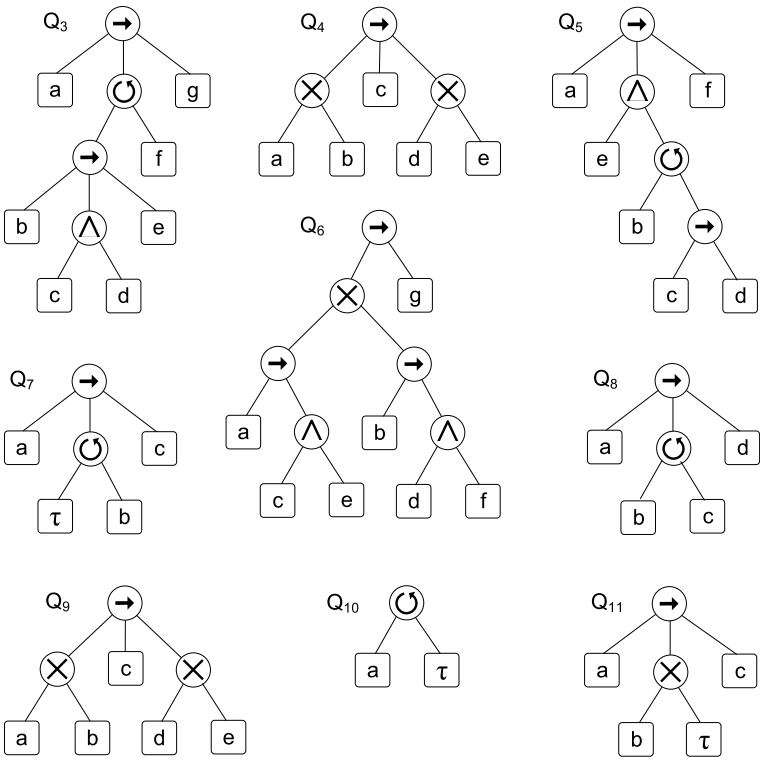

Fig. 7.29 Process trees learned for event logs L_3–L_{11} introduced in Chap. 6

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

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

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

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

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

$$L_{10} = \left[\langle a, a\rangle^{55}\right]$$

$$L_{11} = \left[\langle a, b, c\rangle^{20}, \langle a, c\rangle^{30}\right]$$

These example logs were used to illustrate the characteristics and limitations of the α-algorithm. Figure 7.29 shows the process trees learned for these event logs using the basic IM algorithm described in Sect. 7.5.1: $Q_3 = \text{IM}(L_3)$, $Q_4 = \text{IM}(L_4)$, etc.

The α-algorithm was able to discover "correct" models for L_1–L_5 where correctness is defined as the ability to reproduce the event log. For L_6 the α-algorithm produces a Petri net with two redundant places, but the discovered model is trace equivalent to the desired model (see Fig. 6.9). The α-algorithm cannot handle the loop of length one required for L_7. Also loops of length two cannot be discovered and hence event log L_8 is not handled well. The α-algorithm creates an underfitting model for L_9. The α-algorithm is also unable to handle the repetition of a in L_{10} and the skipping of b in L_{11}.

Figure 7.29 shows the process trees generated by the IM algorithm for L_3–L_{11}. Unlike the α-algorithm, all models are by definition sound and can replay the respective event log (i.e., perfect fitness). Hence, Q_1–Q_{11} are "correct" in the sense mentioned before. Whereas the α-algorithm was unable to handle short loops (length one or length two), the IM algorithm creates Q_7 and Q_8 illustrating that loops pose no problem. Process tree Q_{11} shows that the skipping of activities can be handled by the IM algorithm. However, process trees Q_9 and Q_{10} also show that the discovered models may be underfitting.

In event log $L_9 = [\langle a, c, d\rangle^{45}, \langle b, c, e\rangle^{42}]$, activity a is eventually followed by d and b is eventually followed by e, but process tree $Q_9 = \rightarrow(\times(a, b), c, \times(d, e))$ does not capture this non-local dependency and allows a to be followed by e. This is not surprising since the process tree representation does not allow for such a non-local dependency (without label splitting). Process tree $Q'_9 = \times(\rightarrow(a, c, d), \rightarrow(b, c, e))$ better captures the behavior seen in L_9, but requires the duplication of activity c. Label duplication would be the best choice here, but often label duplication leads to overfitting models simply enumerating parts of the event log.

The basic IM algorithm also cannot handle repetitions of a fixed length. Process tree $Q_{10} = \circlearrowleft(a, \tau)$ is discovered for event log $L_{10} = [\langle a, a\rangle^{55}]$. Hence, Q_{10} also allows for unobserved traces like $\langle a\rangle$ and $\langle a, a, a, a\rangle$. Process tree $Q'_{10} = \rightarrow(a, a)$ better captures the behavior seen in L_{10}, but requires the duplication of activity a.

The examples in Fig. 7.29 illustrate the characteristics of the IM algorithm. The algorithm *always* produces a process tree which is sound by construction and able to replay *all* behavior seen (perfect fitness). However, process trees constructed by the IM algorithm may be underfitting if the observed behavior requires a process tree with duplicate or silent activities. Different processes may have the same directly-follows graph, e.g., $\circlearrowleft(\wedge(a, b), \wedge(c, d))$ and $\wedge(\circlearrowleft(a, c), \circlearrowleft(b, d))$ (see Fig. 7.28). Also $\rightarrow(\times(a, b), c, \times(d, e))$ and $\times(\rightarrow(a, c, d), \rightarrow(b, c, e))$ have the same directly-follows graph. Such processes cannot be distinguished by the IM algorithm.

7.5.3 Extensions and Scalability

The basic IM algorithm described in Sect. 7.5.1 was introduced only recently (in 2013) [88]. Yet, several extensions and refinements have been proposed [89–91]. The basic algorithm cannot abstract from infrequent behavior and does not handle incompleteness well. The log is assumed to be directly-follows complete and frequencies are not taken into account. Fortunately, the IM framework is quite flexible.

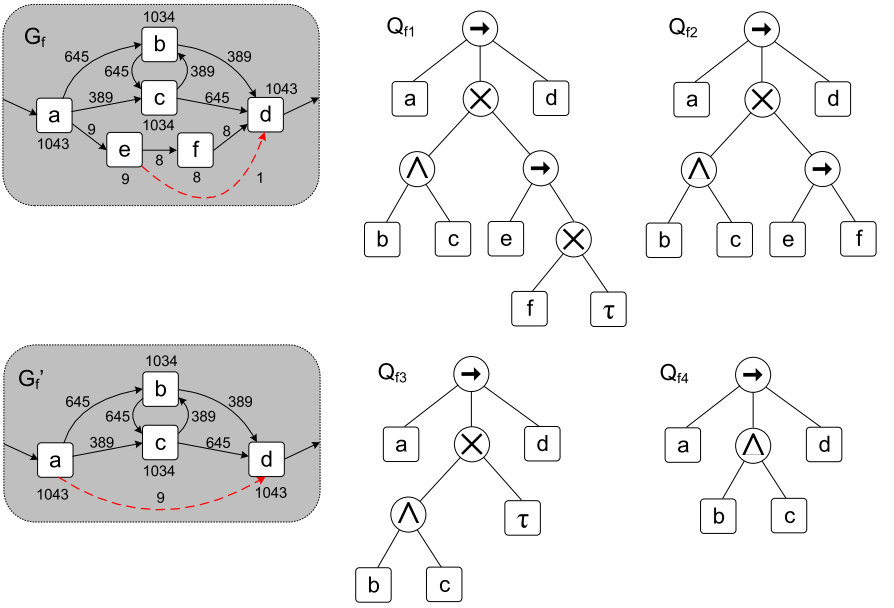

Fig. 7.30 Process trees Q_{f1}–Q_{f4} learned for event log $L_f = [\langle a, b, c, d\rangle^{645}, \langle a, c, b, d\rangle^{389}, \langle a, e, f, d\rangle^{8}, \langle a, e, d\rangle]$ using frequency-based filtering. G_f is the directly-follows graph for the whole log. G'_f is the directly-follows graph after filtering based on activity frequency. The *dashed lines* indicate directly-follows relations that are less frequent and candidates for filtering

Using the basic ideas presented thus far a *family of inductive mining techniques* has been developed. Example members of this family of process discovery algorithms are:

- *Inductive Miner—infrequent* (IMF, [89]),
- *Inductive Miner—incompleteness* (IMC, [90]),
- *Inductive Miner—directly-follows based* (IMD, [91]),
- *Inductive Miner—infrequent—directly-follows based* (IMFD, [91]), and
- *Inductive Miner—incompleteness—directly-follows based* (IMCD, [91]).

To illustrate the IMF (Inductive Miner - infrequent) algorithm consider event log $L_f = [\langle a, b, c, d\rangle^{645}, \langle a, c, b, d\rangle^{389}, \langle a, e, f, d\rangle^{8}, \langle a, e, d\rangle]$. Figure 7.30 shows the directly-follows graph $G_f = G(L_f)$ based on event log L_f. The numbers indicate frequencies, e.g., activity b was executed 1034 times and was directly followed by activity c 645 times. The arc between e and d is dashed because this directly-follows relation is infrequent compared to all other arcs (the only "witness" is trace $\langle a, e, d\rangle$ that occurs once). Filtering out this arc would yield process tree Q_{f2} rather than Q_{f1}. In Q_{f2}, activity f cannot be skipped, this option was removed because it happens only once in L_f. Next to filtering arcs, it is also possible to filter activities. Activities e and f occur less frequent than the other activities. It is possible to remove these activities from the event log resulting in event log $L'_f = [\langle a, b, c, d\rangle^{645},$

$\langle a, c, b, d\rangle^{389}$, $\langle a, d\rangle^{9}$]. Figure 7.30 shows the directly-follows graph $G'_f = G(L'_f)$ based on the filtered event log. Process tree Q_{f3} is based on this directly-follows graph. The arc between a and d is dashed because this directly-follows relation is infrequent (only 9 times a is followed by d). Filtering out this arc would yield process tree Q_{f4} rather than Q_{f3}. The IMF algorithm uses various types of filtering with the goal to show the mainstream behavior. Figure 7.30 only sketches the basic principles. The IMF algorithm is more sophisticated and also uses the "eventually-follows graph" for filtering (next to the directly-follows graph). Note that there are some similarities with the heuristic miner. See [89] for details.

The IMC (Inductive Miner—incompleteness) algorithm [90] complements the IMF algorithm. Instead of removing exceptional behavior, the problem of missing behavior due to the incompleteness of the event log is addressed. The assumption that event logs are directly-follows complete is unrealistic for less structured processes and relatively small event logs. Consider $Q_c = \wedge(a_1, a_2, \ldots, a_{10})$ and some event log L_c generated from this process tree. There are $10! = 3{,}628{,}800$ possible interleavings in Q_c. Suppose we have an event log with 500 cases. Obviously, this event log only shows a fraction of the possible interleavings (less than 1 out of 7000). Since the different interleavings have different probabilities (e.g., due to different delay distributions), it may also be the case that not all of the 90 possible directly-follows relations appear in L_c. Hence, arcs may be missing in the directly-follows graph. This makes it impossible to apply the parallel cut $(\wedge, \{a_1\}, \{a_2\}, \ldots, \{a_{10}\})$. As a result the fall-through described before needs to used $(\circlearrowleft(\tau, a_1, \ldots))$, resulting in an underfitting model. The IMC algorithm uses so-called "probabilistic activity relations" [90] based on both the directly-follows graph and the eventually-follows graph. These are used to select the "most likely cut" even if the requirements stated in Definition 7.6 are not fully satisfied.

The IM, IMF, and IMC algorithms perform quite well compared to other algorithms (e.g., much faster than region-based techniques). However, the event log needs to be split recursively. This may create quite some overhead for larger event logs. Ideally, a single pass through the event log is preferable from a performance point of view. However, the IM, IMF, and IMC algorithms repeatedly traverse the event log to create smaller logs.

The *Inductive Miner—directly-follows based* (IMD) *framework* recurses on the directly-follows graph directly without creating sublogs [91]. This makes the framework extremely scalable. A single pass through the event log suffices and the work can be distributed easily. However, there are some limitations related to the accuracy of the results. There exist variants of the IM, IMF, and IMC algorithms using this framework. These are called the IMD (Inductive Miner—directly-follows based) algorithm, the IMFD (Inductive Miner—infrequent—directly-follows based) algorithm, and the IMCD (Inductive Miner—incompleteness—directly-follows based) algorithm. The cut detection works as before. However, the directly-follows graph is split into disjoint subgraphs (the graphs are not recomputed over sublogs).

The IMD algorithm runs in $O(n^3)$ where n is the number of activities in the directly-follows graph [91]. However, the guarantees provided by the IMD algorithm are similar to the basic IM algorithm. Still most process trees without dupli-

cate and silent activities are language-rediscoverable. If the event log is directly-follows complete and situations such as the one shown in Fig. 7.28 are excluded, then the IMD algorithm is able to rediscover the model used to generate the event log. This only holds under the assumption that there are no duplicate and silent activities.

The IMD framework also has some limitations. The model returned by the IMD algorithm is no longer fitness preserving. Consider directly-follows complete event logs L_1 and L_2 for process trees $Q_1 = \wedge(\rightarrow(a, b), c)$ and $Q_2 = \times(\rightarrow(a, c, b, c, a, b), c)$. The two logs have identical directly-follows graphs, $G(L_1) = G(L_2)$. The IMD algorithm returns Q_1. However, Q_1 cannot reproduce any trace in L_2. Hence, IMD is not fitness preserving for L_2. If the IMD algorithm would not return Q_1, then there would be no event log for which Q_1 could be constructed. This would be undesirable given the prevalence of Q_1's behavior. Hence, fitness preservation is impossible in this setting (without using a fall-through).

The basic IM algorithm that recursively splits the event log is able to distinguish between L_1 and L_2. The IM algorithm rediscovers Q_1: $\text{IM}(L_1) = \wedge(\rightarrow(a, b), c)$. Q_2 is not rediscovered: $\text{IM}(L_2) = \wedge(\circlearrowleft(\tau, \rightarrow(a, b)), \circlearrowleft(c, \tau))$. However, unlike the IMD algorithm the log-splitting IM algorithm guarantees fitness preservation: $\mathscr{L}(Q_2) \subseteq \mathscr{L}(\text{IM}(L_2))$.

The limitations of the IMD framework are counterbalanced by its remarkable *scalability*. The IMD algorithm can handle event logs with billions of events while using only 2 GB of RAM [91]. It can be used to learn process models with over 10,000 activities. Moreover, computation can be easily distributed (e.g., using the Map-Reduce programming model and Hadoop-like infrastructures, see Chap. 12).

The family of inductive mining techniques also includes approaches that take into account transactional information (e.g., start and complete). The basic idea of all algorithms is to use a *divide-and-conquer* approach in combination with process trees that are *sound by construction*.

The different inductive mining algorithms (IM, IMF, IMC, IMD, IMFD, IMCD, etc.) combine interesting properties. The produced models are always sound. The algorithms are highly scalable, in particular IMD and IMFD. If desired, the algorithms are fitness-preserving (i.e., the log can be reproduced by models discovered using IM or IMC). Moreover, models can be seamlessly simplified by leaving out infrequent behavior (IMF and IMFD) and even event logs that are not directly-follows complete can be handled (IMC and IMCD). For particular classes of models even rediscoverability is guaranteed (IM and IMD). Trade-offs between scalability, accuracy, generalization, and precision are supported. These characteristics make inductive mining the current frontrunner in process discovery.

7.6 Historical Perspective

On the one hand, process mining is a relatively young field. All the process discovery techniques described in this chapter were developed in the last decade. More-

over, it is only recently that mature process discovery techniques and effective implementations have become available. On the other hand, process discovery has its roots in various established scientific disciplines ranging from concurrency theory, inductive inference and stochastics to data mining, machine learning and computational intelligence. It is impossible to do justice to the numerous contributions to process mining originating from different scientific domains. Hence, this section should be seen as a modest attempt to provide a historical perspective on the origins of process discovery.

In 1967 Mark Gold showed in his seminal paper "Language identification in limit" [61] that even regular languages cannot be exactly identified from positive examples only. In [61] Gold describes several inductive inference problems. The challenge is to guess a "rule" (e.g., a regular expression) based on an infinite stream of examples. An inductive inference method is able to "learn the rule in the limit" if after a finite number of examples the method is always able to guess the correct rule and does not need to revise its guess anymore based on new examples. A regular language is a language that can be accepted by a finite transition system (also referred to as a finite state machine). Regular languages can also be described in terms of regular expressions. For example, the regular expression $ab^*(c|d)$ denotes the set of traces starting with a, then zero or more b's and finally a c or d. Regular expressions were introduced by Stephen Cole Kleene [85] in 1956. In the Chomsky hierarchy of formal grammars, regular languages are the least expressive (i.e., Type-3 grammar). For example, it is impossible to express the language $\{a^n b^n \mid n \in \mathbb{N}\}$, i.e., the language containing traces that start with any number of a's followed by the same number of b's. Despite the limited expressiveness of regular expressions, Gold showed in [61] that they *cannot* be learned in the limit from positive examples only.

Many inductive inference problems have been studied since Gold's paper (see the survey in [15]). For instance, subclasses of the class of regular languages have been identified that can be learned in the limit (e.g., the so-called k-reversible languages [15]). Moreover, if an "oracle" is used that can indicate whether particular examples are possible or not, a larger class of languages can be learned. This illustrates the importance of negative examples when learning. However, as indicated before, one will not find negative examples in an event log; *the fact that something did not happen provides no guarantee that it cannot happen*. Inductive inference focuses on learning a language perfectly. This is not the aim of process mining. Real-life event logs will contain noise and are far from complete. Therefore, the theoretical considerations in the context of inductive inference are less relevant for process mining.

Before the paper of Gold, there were already techniques to construct a finite state machine from a finite set of example traces. A naïve approach is to use the state representation function $l_1^{state}(\sigma, k) = hd^k(\sigma)$ described in Sect. 7.4.1 to construct a finite state machine. Such a finite state machine can be made smaller by using the classical Myhill–Nerode theorem [108]. Let L be a language over some alphabet $\mathscr{A}$ and consider $\sigma_x, \sigma_y \in \mathscr{A}^*$. σ_x and σ_y are *equivalent* if there is no $\sigma_z \in \mathscr{A}^*$ such that $\sigma_x \oplus \sigma_z \in L$ while $\sigma_y \oplus \sigma_z \notin L$ or $\sigma_y \oplus \sigma_z \in L$ while $\sigma_x \oplus \sigma_z \notin L$. Hence, two traces

are equivalent if their "sets of possible futures" coincide. This equivalence notion divides the elements of L into equivalence classes. If L is a regular language, then there are finitely many equivalence classes. The Myhill–Nerode theorem states that if there are k such equivalence classes, then the smallest finite state machine accepting L has k states. Several approaches have been proposed to minimize finite state machines using these insights (basically folding equivalent states). In [21], a modification of the Myhill–Nerode equivalence relation is proposed for constructing a finite state machine based on a set of sample traces L with a parameter to balance precision and complexity. Here two states are considered equivalent if their k-tails are the same. In 1972, Alan Biermann also proposed an approach to "learn" a Turing machine from a set of sample computations [20].

In the mid 1990s, people like Rakesh Agrawal and others developed various data mining algorithms to find frequent patterns in large datasets. In [7], the Apriori algorithm for finding association rules was presented. These techniques were extended to sequences and episodes [69, 94, 131]. However, none of these techniques aimed at discovering end-to-end processes. More related is the work on hidden Markov models [9]. Here end-to-end processes can be considered. However, these models are sequential and cannot be easily converted into readable business process models.

In the second half of the 1990s, Cook and Wolf developed process discovery techniques in the context of software engineering processes. In [33], they described three methods for process discovery: one using neural networks, one using a purely algorithmic approach, and one Markovian approach. The authors considered the latter two to be the most promising approaches. The purely algorithmic approach builds a finite state machine in which states are fused if their futures (in terms of possible behavior in the next k steps) are identical. (Note that this is essentially the approach proposed by Biermann and Feldmann in [21].) The Markovian approach uses a mixture of algorithmic and statistical methods and is able to deal with noise. All approaches described in [33] are limited to sequential processes, i.e., no concurrency is discovered.

In 1998, two papers [8, 38] appeared that, independently of one another, proposed to apply process discovery in the context of business process management.

In [8], Agrawal, Gunopulos, and Leymann presented an approach to discover the so-called "conformal process graph" from event logs. This work was inspired by the process notation used by Flowmark and the presence of event logs in WFM systems. The approach discovers causal dependencies between activities, but is not able to find AND/XOR/OR-splits and joins, i.e., the process logic is implicit. Moreover, the approach has problems dealing with loops: a trace $\langle a, a, a \rangle$ is simply relabeled into $\langle a_1, a_2, a_3 \rangle$ to make the conformal process graph acyclic.

In the same year, Anindya Datta [38] proposed a technique to discover business process models by adapting the Biermann–Feldmann algorithm [21] for constructing finite state machines based on example traces. Datta added probabilistic elements to the original approach and embedded the work in the context of workflow management and business process redesign. The approach assumes that case identifiers are unknown, i.e., the setting is similar to the work in [53] where the challenge is to correlate events and discover cases. The resulting process model is again a sequential model.

Joachim Herbst [71, 72] was one of the first aiming at the discovery of more complicated process models. He proposed stochastic task graphs as an intermediate representation before constructing a workflow model in terms of the ADONIS modeling language. In the induction step, task nodes are merged and split in order to discover the underlying process. A notable difference with most approaches is that the same activity can appear multiple times in the process model, i.e., the approach allows for duplicate labels. The graph generation technique is similar to the approach of [8]. The nature of splits and joins (i.e., AND or OR) is discovered in the transformation step, i.e., the step in which the stochastic task graph is transformed into an ADONIS workflow model with block-structured splits and joins.

Most of the classical approaches have problems dealing with concurrency, i.e., either sequential models are assumed (e.g., transition systems, finite state machines, Markov chains, and hidden Markov models) or there is a post-processing step to discover concurrency. The first model to adequately capture concurrency was already introduced by Carl Adam Petri in 1962 [111]. (Note that the graphical notation as we know it today was introduced later.) However, classical process discovery techniques do not take concurrency into account. The α-algorithm [157] described in Sect. 6.2 and a predecessor of the heuristic miner [184] described in Sect. 7.2 were developed concurrently and share the same ideas when it comes to handling concurrency. These were the first process discovery techniques taking concurrency as a starting point (and not as an afterthought or post-optimization). The α-algorithm was used to explore the theoretical limits of process discovery [157]. Several variants of the α-algorithm have been proposed to lift some of its limitations [10, 11, 171, 174, 185]. The focus of heuristic mining was (and still is) on dealing with noise and incompleteness [183, 184].

Techniques such as the α-algorithm and heuristic mining do not guarantee that the model can replay all cases in the event log. In [171, 172], an approach is presented that guarantees a fitness of 1, i.e., all traces in the event log can be replayed in the discovered model. This is achieved by creatively using OR-splits and joins. As a result, the discovered model is typically underfitting. In [59, 60], artificially generated "negative events" are inserted to transform process discovery into a classification problem. The insertion of negative events corresponds to the completeness assumptions made by algorithms like the α-algorithm, e.g., "if a is never directly followed by b, then this is not possible".

Region-based approaches are able to express more complex control-flow structures without underfitting. State-based regions were introduced by Ehrenfeucht and Rozenberg [51] in 1989 and generalized by Cortadella et al. [34]. In [165, 173], it is shown how these state-based regions can be applied to process mining. In parallel, several authors applied language-based regions to process mining [19, 170]. In [28], Joseph Carmona and Jordi Cortadella present an approach based on convex polyhedra. Here, the Parikh vector of each prefix in the log is seen as a polyhedron. By taking the convex hull of these convex polyhedra one obtains an over-approximation of the possible behavior. The resulting polyhedron can be converted into places using a construction similar to language-based regions. The synthesis/region-based approaches typically guarantee a fitness of 1. Unfortunately, these approaches also have problems dealing with noise.

For practical applications of process discovery it is essential that noise and incompleteness are handled well. Surprisingly, only few discovery algorithms described in literature focus on addressing these issues. Notable exceptions are heuristic mining [183, 184], fuzzy mining [66], genetic process mining [12, 26], and inductive mining [89–91]. Therefore, we put emphasis on these techniques in this chapter.

See [156, 160, 174] for additional pointers to earlier related work. These surveys do not include recent developments such as the family of inductive mining techniques presented in Sect. 7.5. The different inductive mining algorithms (IM, IMF, IMC, IMD, IMFD, IMCD, etc.) always produce sound models and are highly scalable. Moreover, these algorithms come with formal guarantees. For example, the IM algorithm is fitness-preserving and for particular classes of models even rediscoverability is guaranteed.