11.4.2 The α-Algorithm

The *α-algorithm* is a basic algorithm for discovering process models from event logs. The algorithm is simple enough that we can easily grasp how it works in detail. However, it makes a strong assumption about the event log, namely *behavioral completeness*. An event log is behaviorally complete if whenever a task a can be directly followed by a task b in the actual process, then there is at least one case in the event log where we observe ab. In practice, we can hardly assume that we find the complete set of behavioral options in a log. Advanced techniques, which we will discuss later, try to lift this assumption by trying to guess which traces the process might have, which are not present in the event log.

The α-algorithm constructs a process model from a behaviorally complete event log in two phases. In the first phase, a set of *order relations* between pairs of events are extracted from the workflow log. In the second phase, the process model is constructed in a stepwise fashion from these identified relations. The *order* relations refer to pairs of tasks which directly follow one another in the log. They provide the basis for the definition of three more specific relations that refer to *causality*, to potential *parallelism* and to *no-direct-succession*. We refer to this set of relations as the α relations.

- The basic directly-follows relation $a > b$ holds if we can observe in the workflow log L that a task a is directly followed by b. This is the same relation as that captured in a dependency graph.
- The causality relation $a \rightarrow b$ is derived from the directly-follows relation. It holds if we observe in L that $a > b$ and that $b \not> a$.
- The relation of potential parallelism $a||b$ holds if both $a > b$ and $b > a$ are observed in the workflow log L.
- The relation of no-direct-succession $a\#b$ holds if $a \not> b$ and $b \not> a$.

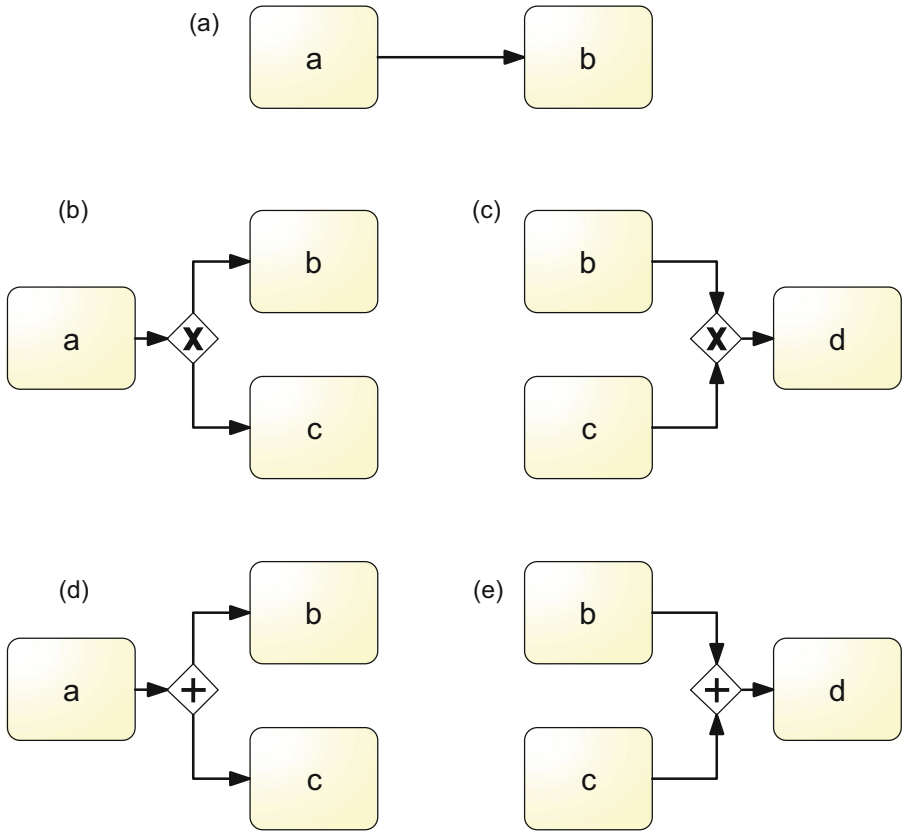

Fig. 11.10 Simple control flow patterns

The reason why exactly these relations are used is illustrated in Figure 11.10. There exist five characteristic combinations of relations between the tasks in a workflow log that can be mapped to simple control flow patterns.

Pattern (a) depicts a sequence of tasks a and b. If we model them in this way, it should be guaranteed that in a workflow log we will find a followed by b, i.e., $a > b$, but never b followed by a, i.e., $b \not> a$. This means that the causality relation $a \rightarrow b$ should hold.

Pattern (b) also relates to a characteristic combination of relations. The workflow log should show that $a \rightarrow b$ and $a \rightarrow c$ hold, and that b and c would not be mutual successors, i.e., $b \# c$.

Pattern (c) also requires that b and c are not mutual successors, i.e., $b \# c$, while both $b \rightarrow d$ and $c \rightarrow d$ have to hold.

Pattern (d) demands that $a \rightarrow b$ and $a \rightarrow c$ hold, and that b and c show potential parallelism, i.e., $b || c$.

Pattern (e) refers to $b \rightarrow d$ and $c \rightarrow d$ while b and c show potential parallelism, i.e., $b || c$.

The idea of the α-algorithm is to identify the relations between all pairs of tasks from the workflow log in order to reconstruct a process model based on Patterns (a) to (e). Therefore, before applying the α-algorithm, we must extract all basic

order relations from the workflow log L. Consider the workflow log depicted in Figure 11.7 containing the two cases $\langle a, b, g, h, j, k, i, l\rangle$ and $\langle a, c, d, e, f, g, j, h, i, k, l\rangle$. From this workflow log, we can derive the following relations.

- The basic order relations $>$ refer to pairs of tasks in which one task directly follows the another. These relations can be directly read from the log:

$a > b$	$h > j$	$i > l$	$d > e$	$g > j$	$i > k$
$b > g$	$j > k$	$a > c$	$e > f$	$j > h$	$k > l$
$g > h$	$k > i$	$c > d$	$f > g$	$h > i$	

- The causal relations can be found when an order relation does not hold in the opposite direction. This is the case for all pairs except (h, j) and (i, k), and their opposites. We get:

$a \to b$	$j \to k$	$a \to c$	$d \to e$	$f \to g$	$h \to i$
$b \to g$	$i \to l$	$c \to d$	$e \to f$	$g \to j$	$k \to l$
$g \to h$					

- The potential parallelism relation holds true for $h||j$ as well as for $k||i$ (and the corresponding symmetric cases).
- The remaining relation of no-direct-succession can be found for all pairs that do not belong to $\to$ and $||$. It can be easily derived when we write down the relations in a matrix as shown in Figure 11.11. This matrix is also referred to as the *footprint matrix* of the log.

Exercise 11.8 Have a look at the workflow log you constructed in Exercise 11.5 (page 427). Define the relations $>, \to, ||, \#$, as well as the footprint matrix of this workflow log.

The α-algorithm takes an event log L and its α relations as a starting point. The essential idea of the algorithm is that whenever a task directly follows another one in the log, the two tasks should be directly connected in the process model. Furthermore, if there is more than one task that can follow another, we have to determine whether the set of succeeding tasks is partially exclusive or concurrent. An exception from the principle that tasks should be connected in the process model is when two tasks are potentially parallel, i.e., those pairs included in $||$. The details of the α-algorithm are defined according to the following eight steps.[17]

1. Let T_L be the set of all tasks in the log.
2. Let T_I be the set of tasks that appear at least once as first task of a case.

[17]Note that the α-algorithm was originally defined for constructing Petri nets. The version shown here is a simplification based on the five simple control-flow patterns of Figure 11.10, in order to construct BPMN models.

	a	b	c	d	e	f	g	h	i	j	k	l
a	#	$\rightarrow$	$\rightarrow$	#	#	#	#	#	#	#	#	#
b	$\leftarrow$	#	#	#	#	#	$\rightarrow$	#	#	#	#	#
c	$\leftarrow$	#	#	$\rightarrow$	#	#	#	#	#	#	#	#
d	#	#	$\leftarrow$	#	$\rightarrow$	#	#	#	#	#	#	#
e	#	#	#	$\leftarrow$	#	$\rightarrow$	#	#	#	#	#	#
f	#	#	#	#	$\leftarrow$	#	$\rightarrow$	#	#	#	#	#
g	#	$\leftarrow$	#	#	#	$\leftarrow$	#	$\rightarrow$	#	$\rightarrow$	#	#
h	#	#	#	#	#	#	$\leftarrow$	#	$\rightarrow$	$\|\|$	#	#
i	#	#	#	#	#	#	#	$\leftarrow$	#	#	$\|\|$	$\rightarrow$
j	#	#	#	#	#	#	$\leftarrow$	$\|\|$	#	#	$\rightarrow$	#
k	#	#	#	#	#	#	#	#	$\|\|$	$\leftarrow$	#	$\rightarrow$
l	#	#	#	#	#	#	#	#	$\leftarrow$	#	$\leftarrow$	#

Fig. 11.11 Footprint represented as a matrix of the workflow log $L = [\langle a, b, g, h, j, k, i, l\rangle, \langle a, c, d, e, f, g, j, h, i, k, l\rangle]$

3. Let T_O be the set of tasks that appear at least once as last task in a case.
4. Let X_L be the set of potential task connections. X_L is composed of:
   a. Pattern (a): all pairs for which $a \rightarrow b$ holds.
   b. Pattern (b): all triples for which $a \rightarrow (b\#c)$ holds.
   c. Pattern (c): all triples for which $(b\#c) \rightarrow d$ holds.

   Note that triples for which Pattern (d) $a \rightarrow (b||c)$ or Pattern (e) $(b||c) \rightarrow d$ hold are not included in X_L.
5. Construct the set Y_L as a subset of X_L by:
   a. Eliminating $a \rightarrow b$ and $a \rightarrow c$ if there exists some $a \rightarrow (b\#c)$.
   b. Eliminating $b \rightarrow c$ and $b \rightarrow d$ if there exists some $(b\#c) \rightarrow d$.
6. Connect start and end events in the following way:
   a. If there are multiple tasks in the set T_I of first tasks, then draw a start event leading to an XOR-split, which connects to every task in T_I. Otherwise, directly connect the start event with the first task.
   b. For each task in the set T_O of last tasks, add an end event and draw an arc from the task to the end event.
7. Construct the flow arcs in the following way:
   a. Pattern (a): for each $a \rightarrow b$ in Y_L, draw an arc a to b.
   b. Pattern (b): for each $a \rightarrow (b\#c)$ in Y_L, draw an arc from a to an XOR-split, and from there to b and c.

c. Pattern (c): for each $(b\#c) \rightarrow d$ in Y_L, draw an arc from b and c to an XOR-join, and from there to d.
d. Pattern (d) and (e): if a task in the so-constructed process model has multiple incoming or multiple outgoing arcs, bundle these arcs with an AND-join or AND-split, respectively.

8. Return the newly constructed process model.

Let us apply the α-algorithm to $L = [\langle a, b, g, h, j, k, i, l\rangle, \langle a, c, d, e, f, g, j, h, i, k, l\rangle]$. Steps 1–3 identify $T_L = \{a, b, c, d, e, f, g, h, i, j, k, l\}$, $T_I = \{a\}$, and $T_O = \{l\}$. In Step 4a all causal relations are added to X_L including $a \rightarrow b$,$a \rightarrow c$, etc. In Step 4b, we work row by row through the footprint matrix of Figure 11.11 and check if there are cells sharing a $\rightarrow$ relation while relating to tasks that are pairwise in $\#$. In the row a, we observe both $a \rightarrow b$ and $a \rightarrow c$. Also, $b\#c$ holds. Therefore, we add $a \rightarrow (b\#c)$ to X_L. We also consider row g and its relation to h and j. However, as $h||j$ holds, we do not add them. In Step 4c, we progress column by column through the footprint matrix and see if there are cells sharing a $\rightarrow$ relation while relating to tasks that are mutually in $\#$. In column g, we observe two $\rightarrow$ relations to b and f. Also, $b\#f$ holds. Accordingly, we add $(b\#f) \rightarrow g$ to X_L. We also check i and k that share the same relation to l. However, as $i||k$ holds, we do not add them. There are no further complex combinations found in Step 4d.

In Step 5, we eliminate the basic elements in X_L that are covered by the complex patterns found in Steps 4b and 4c. Accordingly, we delete $a \rightarrow b$, $a \rightarrow c$, $b \rightarrow g$, and $f \rightarrow g$. In Step 6a we introduce a start event and connect it with a; in 6b, task l is connected with an end event. In Step 7, arcs and gateways are added for the elements of Y_L. Finally, in Step 8 the resulting process model is returned, as shown in Figure 11.12.

Exercise 11.9 Consider the workflow log and the footprint you constructed in Exercises 11.5 (page 427) and 11.8 (page 434). Show step-by-step how the α-algorithm works on this workflow log, and draw the resulting process model.

11.4.3 Robust Process Discovery

The α-algorithm can reconstruct a process model from a behaviorally complete event log if that log has been generated from a structured process model. There are also limitations to be noted, though. The α-algorithm is not able to distinguish so-called *short loops* from true parallelism. As can be seen in Figure 11.13, all three models can produce a workflow log that yields $b||c$ in the corresponding footprint. Several extensions to the α-algorithm have been proposed. The idea of the $\alpha+$-algorithm is to define the relation $||$ in a stricter way such that $b||c$ is only included if there is no sequence bcb in the log. In this way, models (b) and (c) in Figure 11.13 can be distinguished from model (a) in their generated logs. Furthermore, we can use pre-processing to extract direct repetitions like aa or bb from a log, note down

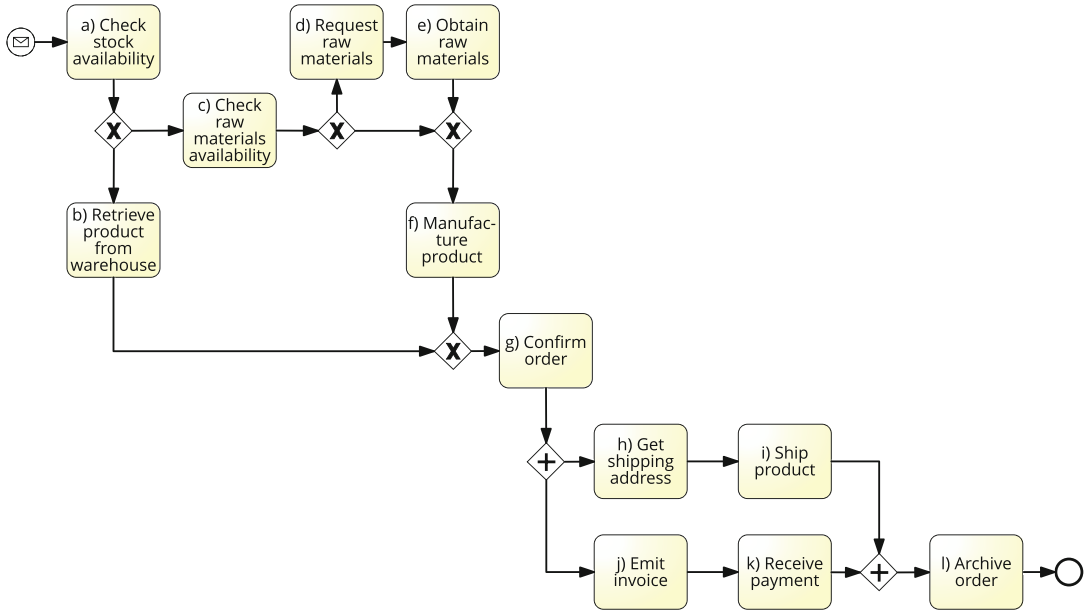

Fig. 11.12 Process model constructed by the α-algorithm from log $L = [\langle a, b, g, h, j, k, i, l\rangle, \langle a, c, d, e, f, g, j, h, i, k, l\rangle]$

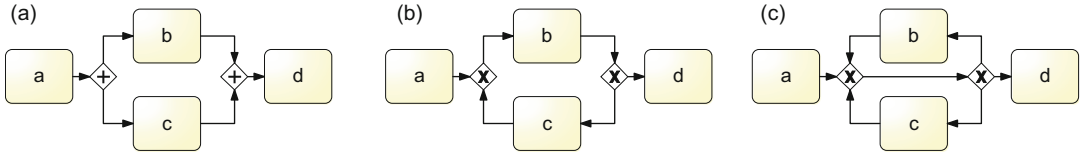

Fig. 11.13 Examples of two short loops, (**b**) and (**c**), that cannot be distinguished from model (**a**) by the α-algorithm

the corresponding tasks, and continue with a log from which such repeated behavior is mapped to a single execution.

Another limitation of the α-algorithm is its inability to deal with *incompleteness* in an event log. The α-algorithm assumes that the $>$ relation (from which the other relations are derived) is complete, meaning that if the process allows task a to be directly followed by task b, then the relation $a > b$ must be observed in at least one trace of the log. This assumption is too strong for processes that have a lot of parallelism. For example, if in a process there is a block of ten concurrent tasks $a_1, \ldots a_{10}$, we need to observe each relation $a_i > a_j$ for each $i, j \in [1..10]$, meaning 100 possible combinations. It suffices that one of these combinations has not been observed in the event log for the α-algorithm to discover the wrong model. This example shows that we need more robust process discovery algorithms, which can smartly infer relations that are not explicitly observed in the event log, but that are somehow implied by what can be observed in the log.

A related limitation is the α-algorithm's inability to deal with *noise*. Event logs often include cases with a missing head, a missing tail, or a missing intermediate episode because some events in a case have not been recorded. For example, a worker might have forgotten to mark an invoice as *paid*, and hence the "Payment received" event is missing in the corresponding trace. Furthermore, there may be

logging errors leading to events being recorded in the wrong order (or with wrong timestamps) or recorded twice. Ideally, such noise should not distort the process model produced by a process discovery technique.

Fortunately, there are other more robust algorithms for automated process discovery, such as the *heuristics miner* [192], the *structured heuristics miner* [11], the *inductive miner* [86] and the *split miner* [10]. These techniques generally work as follows. First, they construct the dependency graph from the event log. Second, they remove some of the nodes and arcs in the process dependency graph in order to deal with noise (infrequent behavior). Finally, they apply a set of heuristic rules to discover split gateways and join gateways in order to turn the filtered dependency graph into a (BPMN) process model.

For example, to handle noise, the heuristics miner [192] uses a relative frequency metric between pairs of event labels, defined as $a \Rightarrow b = \left(\frac{|a>b|-|b>a|}{|a>b|+|b>a|+1}\right)$, where $|a > b|$ is the number of times when task a is directly followed by task b in the log. This metric has a value close to +1 when task a is directly followed by task b sometimes and task b is (almost) never directly followed by a. This means that there is a clear direct-precedence relation from a to b. When the value of $a \Rightarrow b$ is not close to 1 (for example when it is less than 0.8), it means that the direct precedence relation is not very clear (it might be noise) and hence it is deleted. Other similar frequency metrics are used to detect self loops and short loops in order to avoid the limitations of the α-algorithm.

Once the dependency graph has been filtered and self and short loops have been identified, the heuristics miner identifies split and join gateways by analyzing both the filtered dependency graph and the traces in the log. Specifically, if task a is directly followed by b and by c, but not by both (i.e., after b we do not observe c or we very rarely observe it), the heuristics miner will put an XOR-split between task a on the one hand, and tasks b and c on the other hand. Meanwhile, if a is generally followed by both b and by c, the heuristics miner will put an AND-split after task a and before b and c. The term *generally* here means that the heuristics miner tolerates some cases where a is not followed by b or by c if this happens infrequently. Join gateways are discovered similarly: An XOR-join is placed between tasks b and c on the one hand, and task d on the other hand, if d is generally preceded by either task b or c but not both, while an AND-join is placed there if d is generally preceded by both b and c.

Unlike the α-algorithm, the heuristics miner is able to detect self-loops and short-loops, and to handle incomplete and noisy event logs. However, when applied to large real-life event logs, the heuristics miner often produces process models that are too large, spaghetti-like and not sound (see Section 5.4.1 for a definition of soundness of process models).

As discussed in Section 5.4.3, a desirable property of process models is that they should be block-structured. Block-structured process models are always sound, and in addition, they are generally easier to understand than unstructured ones. Accordingly, some of the most robust process discovery techniques try to produce block-structured process models. This is the case for example of the inductive

miner and the structured heuristics miner. Like the heuristics miner, the inductive miner starts by constructing a dependency graph and filtering out infrequent arcs. But instead of immediately proceeding to identifying gateways from the filtered dependency graph, it first analyzes the filtered dependency graph in order to identify arcs that (if removed) would bisect the dependency graph into two separate parts. By doing this several times, it ends up dividing the dependency graph into blocks. Finally, it discovers the gateways locally within each block, in such a way that each split gateway has a corresponding join gateway of the same type, which is the key characteristic of block-structured process models.

The structured heuristics miner implements an alternative approach. It first applies the heuristics miner in order to discover a process model. This process model might be relatively complex (high number of gateways) and highly unstructured. To improve this model, the structured heuristics miner then applies an algorithm that converts unstructured process models into block-structured ones. In doing so, the algorithm also ensures that the resulting model is sound.

The split miner goes beyond the inductive miner and the structured heuristics miner by discovering process models that are sound, but not always perfectly block-structured. By allowing itself to discover non-block-structured models, the split miner technique discovers higher-quality models as discussed below.

The heuristics and inductive miners can be both found in ProM, while these two algorithms, as well as the structured heuristics miner and the split miner, can all be found in Apromore.