

CHAPTER 3

PROCESS DISCOVERY AND CONFORMANCE CHECKING

1. introduction :

Process discovery is one of the most challenging process mining tasks. Based on an event log a process model is constructed thus capturing the behavior seen in the log. This chapter introduces the topic using the rather naïve α -algorithm and other algorithms .

The alpha miner algorithm nicely illustrates some of the general ideas used by many process mining algorithms and helps to understand the notion of process discovery. Moreover, the α -algorithm serves as a stepping stone for discussing challenges related to process discovery.

2.Process discovery :

Even if process models are designed to the best of the designer's capabilities, it is still not guaranteed that they indeed correctly model the process under consideration. This heavily depends on whether they fit to what the people involved in the process want. Furthermore, if the models are enforced during execution, then they are the specification of the operational process and hence they model it correctly. For this purpose, the research topic of process discovery focuses on using event logs to extract information about the operational process. Process discovery is typically applied to gain insights into the operational process, for example to monitor patient flows in hospitals, or the routing of complex products through the different manufacturing stages. Process mining supports the manager or process analyst in finding out what actually happened in the organization. Furthermore, the discovered process models can be used to support a process designer during process design or redesign (i.e. in the design phase).

In contrast to log-based verification, the goal of process discovery is to derive some sort of model that describes the process as accurately as possible. In that respect, we could argue that process discovery focuses on the process perspective of process mining. However, process discovery techniques can be applied from all perspectives, e.g. constructing decision trees for the case perspective and social networks for the organizational perspective.

3.Alpha miner

Input for the α -algorithm is a simple event log L over A , i.e., $L \in B(A^*)$. In the

remainder, we will simply refer to L as the event log. We refer to the elements of A as activities. These activities will correspond to transitions in the discovered Petri net. In this chapter, we will use the convention that capital letters refer to sets of activities (e.g., $A, B \subseteq A$), whereas for individual activities no capitalization is used (e.g., $a, b, c, \dots \in A$). The output of the α -algorithm is a marked Petri net, i.e., $\alpha(L) = (N, M)$. We aim at the discovery of WF-nets. Therefore, we can omit the initial marking and write $\alpha(L) = N$ (the initial marking is implied; $M = [i]$).[2]

	a	b	c	d	e
a	#L1	-> L1	-> L1	# L1	-> L1
b	<- L1	# L1	L1	-> L1	# L1
c	<- L1	L1	# L1	-> L1	# L1
d	# L1	<- L1	<- L1	# L1	<- L1
e	<- L1	# L1	# L1	-> L1	# L1

Table 3.1 : Footprint of L1: $a \# L1$ a, $a \rightarrow L1$ b, $a \rightarrow L1$ c, etc

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

Definition 3.1 : (Log-based ordering relations) Let L be an event log over A , i.e.,

$L \in B(A^*)$. Let $a, b \in A$.

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

Consider, for instance, $L1 = [\langle a, b, c, d \rangle^3, \langle a, c, b, d \rangle^2, \langle a, e, d \rangle]$ again. For this event log, the following log-based ordering relations can be found:

$>_{L1} = (a, b), (a, c), (a, e), (b, c), (c, b), (b, d), (c, d), (e, d)$

$\rightarrow_{L1} = (a, b), (a, c), (a, e), (b, d), (c, d), (e, d)$

$\#_{L1} = (a, a), (a, d), (b, b), (b, e), (c, c), (c, e), (d, a), (d, d), (e, b), (e, c), (e, e)$

$L1 = (b,c),(c,b)$ Relation $>L1$ contains all pairs of activities in a “directly follows” relation. $c >L1 d$ because d directly follows c in trace a,b,c,d . However, $d \not>L1 c$ because c never directly follows d in any trace in the log. $\rightarrow L1$ contains all pairs of activities in a “causality” relation, e.g., $c \rightarrow L1 d$ because sometimes d directly follows c and never the other way around ($c >L1 d$ and $d \not>L1 c$). $bL1c$ because $b >L1 c$ and $c >L1 b$, i.e., sometimes c follows b and sometimes the other way around. $b \not L1 e$ because $b \not>L1 e$ and $e \not>L1 b$. [2]

For any log L over A and $x,y \in A$, $x \rightarrow_L y$, $y \rightarrow_L x$, $x \#_L y$, or xL_y , i.e., precisely one of these relations holds for any pair of activities. Therefore, the footprint of a log can be captured in a matrix as shown in Table 3.1. The footprint of event log $L2$ is shown in Table 2. The subscripts have been removed to not clutter the table. When comparing the footprints of $L1$ and $L2$ one can see that only the e and f columns and rows differ.[1]

	a	b	c	d	e	f
a	#	->	->	#	#	#
b	<-	#		->	->	<-
c	<-		#	->	->	<-
d	#	<-	<-	#	#	#
e	#	<-	<-	#	#	->
f	#	->	->	#	<-	#

Table 3.2 FootprinofL

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

The log-based ordering relations can be used to discover patterns in the corresponding process model as is illustrated in Fig3.1 . If a and b are in sequence, the log will show $a \rightarrow_L b$. If after a there is a choice between b and c , the log will show $a \rightarrow_L b$, $a \rightarrow_L c$, and $b \#_L c$ because a can be followed by b and c , but b will not be followed by c and vice versa. The logical counterpart of this so-called XOR-split pattern is the XOR-join pattern as shown in Fig 1 (b)–(c). If $a \rightarrow_L c$, $b \rightarrow_L c$, and $a \#_L b$, then this suggests that after the occurrence of either a or b , c should happen. Figure 3.1 (d)–(e) shows the so-called AND-split and AND-join patterns. If $a \rightarrow_L b$, $a \rightarrow_L c$, and bL_c , then it appears that after a both b and c can be executed in parallel (AND-split pattern). If $a \rightarrow_L c$, $b \rightarrow_L c$, and aL_b , then the log suggests that c needs to synchronize a and b (AND-join pattern). Figure 3.1 only shows simple patterns and does not

present the additional conditions needed to extract the patterns. However, the figure nicely illustrates the basic idea.

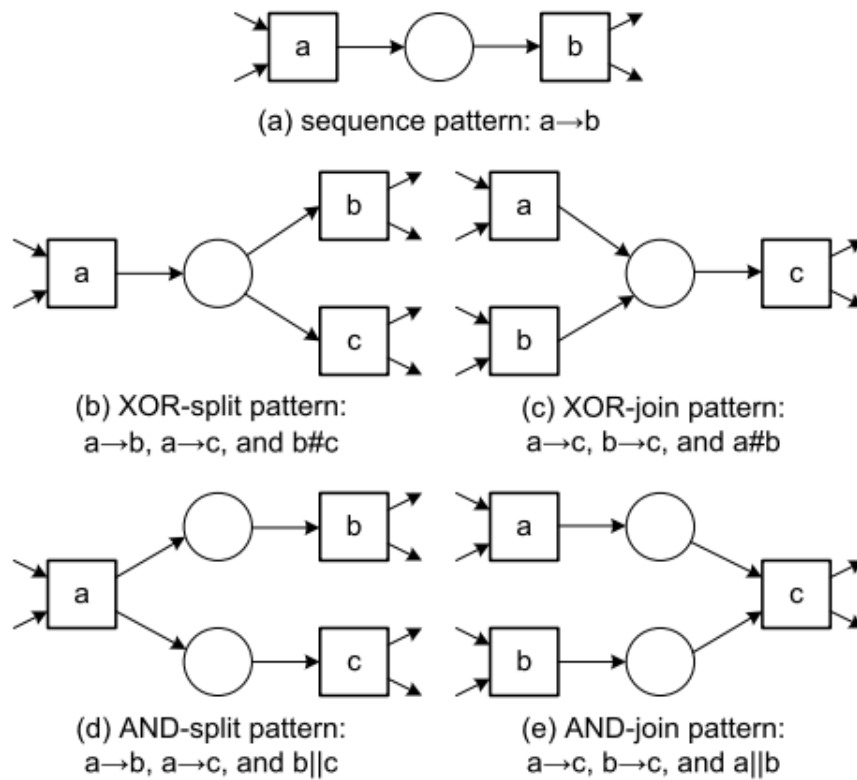


Fig3.1 Typical process patterns and the footprints they leave in the event log [2]

4.Challenges in Process Discovery :

The α -algorithm was the first process discovery algorithm to adequately capture concurrency. Today there are much better algorithms that overcome the weaknesses of the α -algorithm. These are either variants of the α -algorithm or completely different approaches (e.g., based on regions or genetic algorithms).

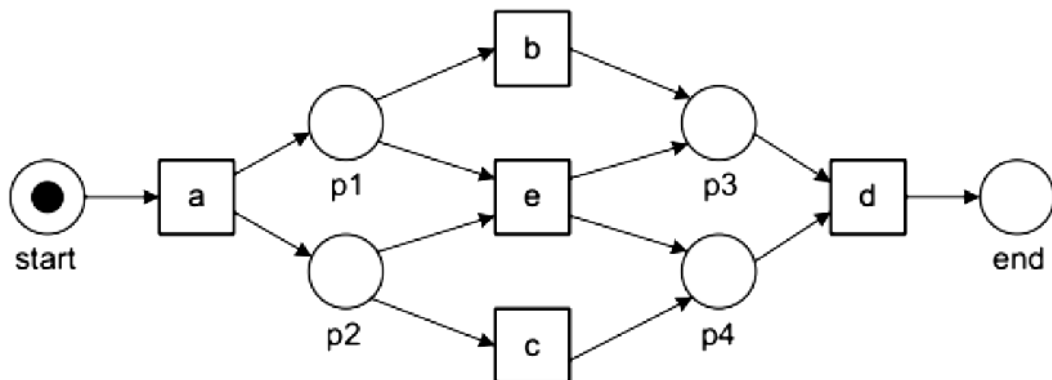


fig 3.2 A workflow net with source place $i = \text{start}$ and sink place $o = \text{end}$ [4]

However, there are still several challenges. Below we list some of these challenges.

The first problem is that many algorithms have problems with complex control-flow constructs. For example, the choice between the concurrent execution of b and c or the execution of just e shown in Figure 3.2 cannot be handled by many algorithms. Most algorithms do not allow for so-called "non-free-choice constructs" where concurrency and choice meet. The concept of free-choice nets is well-defined in the Petri net domain. However, in reality processes tend to be non-free-choice. In the example of Figure 3.2, the α -algorithm is able to deal with the non-free-choice construct. However, it is easy to think of a non-free-choice process that cannot be discovered by the α -algorithm.

The non-free-choice construct is just one of many constructs that existing process mining algorithms have problems with. Other examples are arbitrary nested loops, cancelation, unbalanced splits and joins, partial synchronization, etc. In this context it is important to note that process mining is, by definition, restricted by the expressive power of the target language, i.e., if a simple or highly informal language is used, process mining is destined to produce less relevant or over-simplified results.[5]

The second problem is the fact that most algorithms have problems with duplicates. The same activity may appear at different places in the process or different activities may be recorded in an indistinguishable manner. Consider for example Figure 3.2 and assume that activities a and d are both recorded as x (or, equivalently, assume that a and d are both replaced by activity x). Hence, an event log could be $L_0 = [hx; b; c; x_5; hx; c; b; x_8; hx; e; x_9]$. Most algorithms will try to map the first and the second x onto the same activity. In some cases this makes sense, e.g., to create loops. However, if the two occurrences of x (i.e., a and d) really play a different role in the process, then algorithms that are unable to separate them will run into all kinds of problems, e.g., the model becomes more difficult or incorrect. Since the duplicate activities have the same "footprint" in the log, most algorithms map these different activities onto a single activity thus making the model incorrect or counter-intuitive.

The third problem is that many algorithms have a tendency to generate inconsistent models. Note that here we do not refer to the relation between the log and the model but to the internal consistency of the model by itself. For example, the α -algorithm may yield models that have deadlocks or livelocks when the log shows certain types of behavior. When using Petri nets as a model to represent processes, an obvious choice is to require the model to be sound (i.e., free of deadlocks and other anomalies). Unfortunately, this can typically only be achieved by severely limiting the expressiveness of the modeling languages (e.g., require it to be block structured).[4]

The fourth and last problem described here is probably the most important problem: Existing algorithms have problems balancing between "overfitting" and "underfitting". Overfitting is the problem that a very specific model is generated while it is obvious that the log only holds example behavior, i.e., the model explains the particular sample log but a next sample log of the same process may produce a completely different process model. Underfitting is the problem that the model over-generalizes the example behavior in the log, i.e., the model allows for very different behaviors from what was seen in the log. The problem of balancing between "overfitting" and "underfitting" is related to the notion of completeness discussed next.[4]

5.ProM Tips — Which Mining Algorithm Should You Use?

Probably the most well-known and popular process mining tool available is ProM, an open source toolkit developed at Eindhoven University of Technology. ProM is a good choice to explore process mining, because it has consistently been at the forefront of that technology¹. If you start up ProM for the first time to try out Process Mining, the number of available plugins (almost 300) can be daunting.

Many people have read about the alpha-algorithm in some paper, or in the ProM tutorial, and just keep using that one. Don't do this. For real-life logs, the alpha-algorithm is almost never the right choice. It won't work. Well, it will give you a result, of course (it always does) — But the result won't be good. So, don't use it. And The 3 recommended Mining Algorithms So, which algorithm *should* you use? I can recommend you to use the following three process discovery plugins in ProM. [8]

5.1 Heuristic miner :

The Heuristic Miner was the second process mining algorithm, closely following the alpha algorithm. It was developed by Dr. Ton Weijters, who used a heuristic approach to address many problems with the alpha algorithm, making this algorithm much more suitable in practice.

- *Output:* Heuristic net
- *When to use it:* When you have real-life data with not too many different events, or when you need a Petri net model for further analysis in ProM

The Heuristic miner (previously Little Thumb) derives XOR and AND connectors from dependency relations. It can abstract from exceptional behavior and noise (by leaving out edges) and, therefore, is also suitable for many real-life logs. One of the advantages is that a Heuristic net can be converted to other types of process models, such as a Petri net for further analysis in ProM.

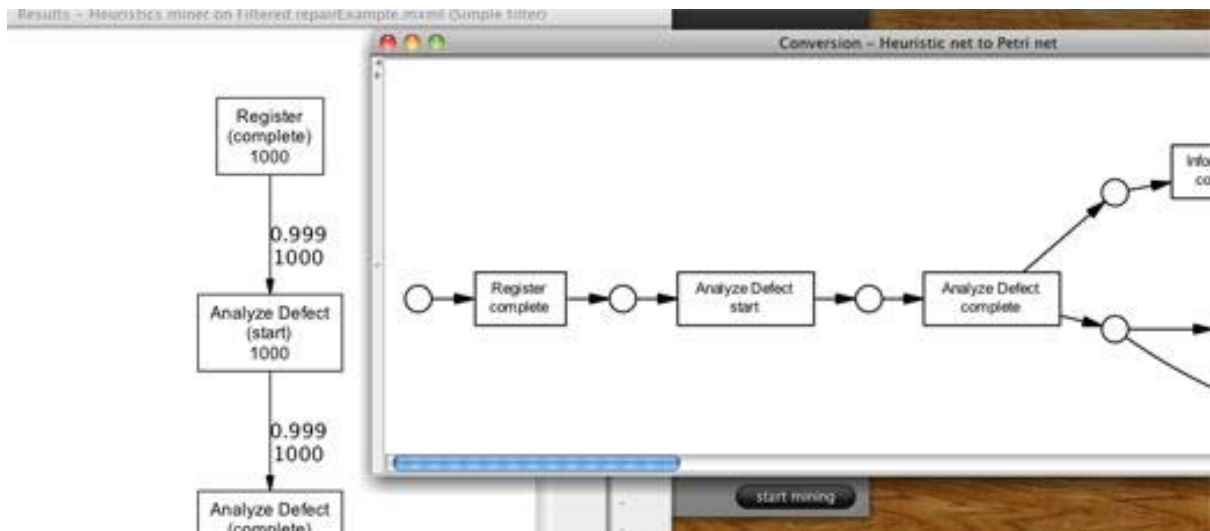


fig 3.3 heuristic miner in prom lite 1.1 [8]

5.2 Fuzzy miner

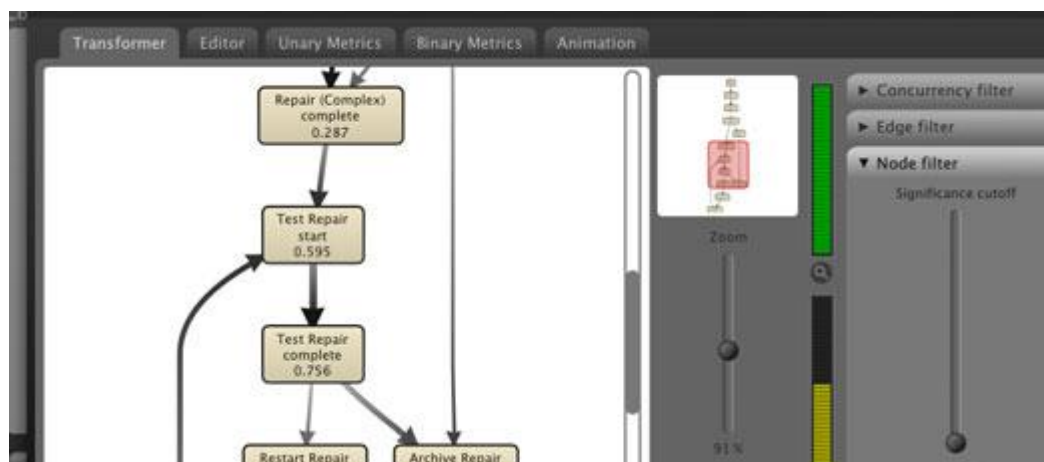


fig 3.4 fuzzy miner in prom lite 1.1 [8]

The Fuzzy miner is one of the younger process discovery algorithms, and was developed by Fluxicon co-founder Christian W. Günther in 2007. It is the first algorithm to directly address the problems of large numbers of activities and highly unstructured behavior.

-Output: Fuzzy Model

-When to use: When you have complex and unstructured log data, or when you want to simplify the model in an interactive manner

The Fuzzy miner uses significance/correlation metrics to interactively simplify the process model at desired level of abstraction. Compared to the Heuristic miner it can also leave out less important activities (or hide them in clusters) if you have hundreds of them. The fuzzy model

cannot be converted to other types of process modeling languages, but you can use it to animate the event log on top of the created model to get a feeling for the dynamic process behavior. [8]

5.3 Multi-phase miner

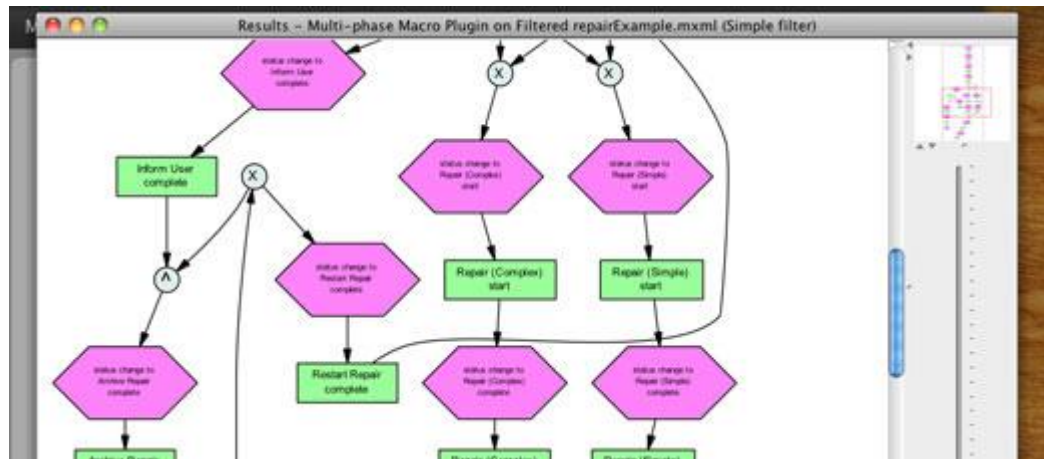


fig 3.5 heuristic miner in prom lite 1.1 [8]

The Multi-phase miner was the first algorithm to explicitly use the OR split/join semantics, as found in EPCs, enabling it to express complex behavior in relatively well-structured models. It was developed by Dr. Boudewijn van Dongen, a process mining veteran and longtime leading developer of ProM.

-*Output*: Event-driven Process Chain (EPC)

-*When to use*: When you have simple and structured log data and you want to export the mining result to Aris

The Multi-phase miner folds XOR, AND, and OR connectors from so-called runs and displays the resulting model as an EPC. The EPC can then be exported to Aris (e.g., in Aris graph format) and further processed from there. One of the advantages of the Multi-phase miner is that it constructs a model that always “fits” the complete event log (more on that in a later post). However, it is seldom useful for more complex processes because the model becomes unreadable.

6. Conformance Checking

The second type of process mining is conformance. Here, an existing process model is compared with an event log of the same process. Conformance checking can be used to check if reality, as recorded in the log, conforms to the model and vice versa.

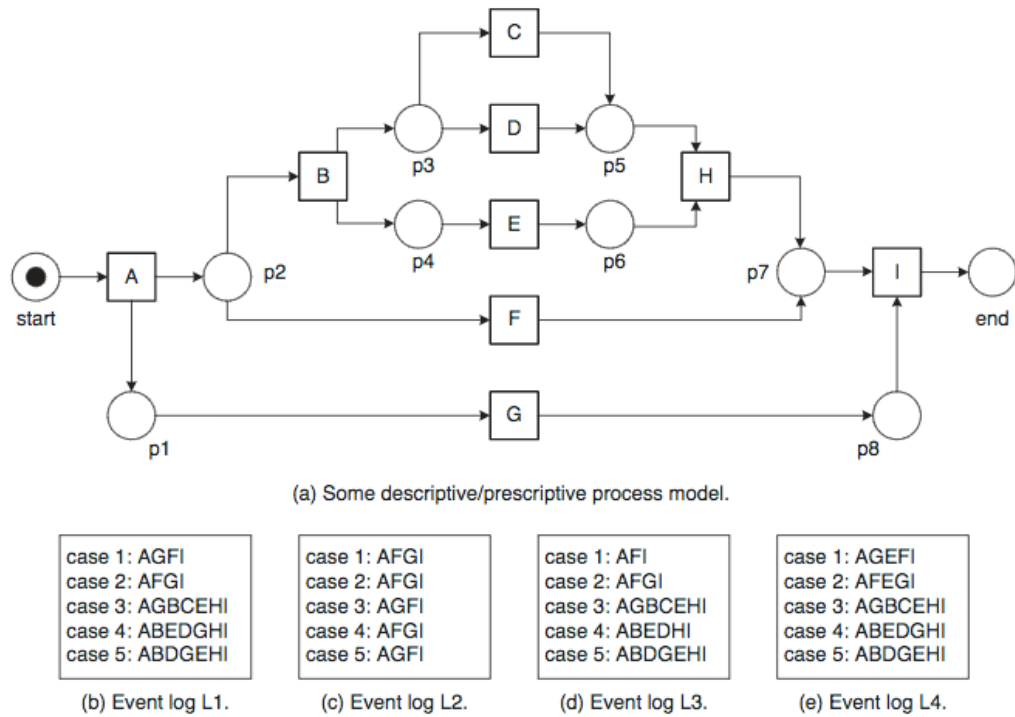


fig3.6 : A predefined process model and some event logs used to illustrate conformance testing.[4]

Consider Figure 3.6 where some predefined model and four logs are given. In the four logs we removed timestamps, performer, etc. and grouped the events in one sequence per case. Event log L1 shows five cases. For the first case (case 1) four events are logged: A, G, F, and I. Clearly this sequence of events is possible in Figure 3.6, i.e., case 1 seems to fit. The same holds for the other four cases in log L1. This can be checked by playing the “token game” in the Petri net shown Figure 3.6. Each case in log L1 corresponds to a possible firing sequence of the Petri net with one initial token in the source place. Similarly, each case in event log L2 corresponds to a possible firing sequence of Figure 3.6. However, it is striking to see that only events corresponding to the lower part of the Petri net appear in the log. Now consider event log L3 shown Figure 3.6. This log does not seem to fit. For example, case 1 does not correspond to a possible firing sequence because after executing A and F, place p8 is still unmarked and I cannot be executed according to the model. It seems that an event involving G is missing. Also case 4 in event log L3 seems to suffer from this problem.[4] Also event log L4 shown Figure 3.6 does not fit completely, i.e., E seems to be impossible in first two cases of this log. Figure 3.6 illustrates that there are two issues when checking the conformance of a predefined process model based on an actual event log. The first problem is underfitting, i.e., the log contains behavior not possible in the predefined model. Event logs L3 and case 4 suffer from underfitting because in each log two cases do not “fit”. The second problem is overfitting, i.e., the log

contains behavior possible in the predefined model but parts of the process model are not addressed by the log. Event log L2 seems to indicate overfitting, i.e., the upper half of the process model is not used according to the log. Given a fixed set of activities X it is easy to construct a Petri net that fits all logs, simply construct the Petri net with one place p and one transition per activity. [3]

Then connect each transition with this p such that p is both an input and an output place. The resulting net fits on any log with just activities of X . Therefore, the predictive value of the resulting Petri net is negligible. Clearly both underfitting and overfitting are highly relevant. However, it is difficult to quantify overfitting. For example, certain paths in the process model may only be used for specific classes of cases (e.g., rush orders) or during specific periods (e.g., during the peak season). The fact that these are not used in some log does not indicate that the process model is necessarily incorrect. However, underfitting can easily be detected as shown by event logs L3 and L4. Therefore, we only consider underfitting in the context of conformance testing. To conclude this section, we consider some metrics for conformance testing. All metrics presented are based on measuring the degree of underfitting. Let P be a process model and L a log. A case c in L can be successfully parsed if and only if c corresponds to a firing sequence of P and after executing this sequence only the sink place is marked (i.e., one token in place end). [5]

The first metric $\text{fit1}(P, L)$ simply counts the number of cases in L that can be parsed by P divided by the number of cases. For example, $\text{fit1}(P, L1) = 1$ while $\text{fit1}(P, L3) = 0.6$ (only three of the five cases can be parsed). This metric does not take into account the number of events. For large process models this may be too simplistic, i.e., the metric does not distinguish between cases that are close to the model and cases that are completely unrelated. Fortunately, it is also possible to simply count the number of events that can be executed.

This can be done using blocking or non-blocking semantics. If we assume blocking semantics, the parsing of the case stops after the first problem is detected. If we use non-blocking semantics, even disabled transitions are fired so that the parsing of the case can continue. Consider for example, case 1 in event log L4. If we use blocking semantics, the parsing stops after executing G because E is not enabled in the Petri net, i.e., only two of the five events can be parsed. If we use non-blocking semantics, the parsing does not stop after executing G and four of the five events can be parsed (all except E). The second metric $\text{fit2}(P, L)$ uses non-blocking semantics to simply count the number of events in L that can be parsed by P divided by the total number of events in the log. Again $\text{fit2}(P, L1) = 1$, but $\text{fit2}(P, L3) = 25/27 = 0.93$ and $\text{fit2}(P, L4) = 25/31 = 0.81$. The third metric $\text{fit3}(P, L)$ uses blocking semantics to count the fraction of

events in L that can be parsed by P . Now $\text{fit}_3(P, L_3) = 25/27 = 0.93$ and $\text{fit}_3(P, L_3) = 27/31 = 0.87$. Clearly, $\text{fit}_3(P, L)$ is a better metric than $\text{fit}_2(P, L)$ because the blocking semantics is very sensitive for the position of the anomaly in the case, i.e., “early problems” outweigh “late problems”.

Note that for $\text{fit}_2(P, L)$ and $\text{fit}_3(P, L)$ we did not specify penalties for tokens remaining in the net after parsing the case. It is possible to define alternative metrics to also take this into account. Table 3.3 summarizes the results for the three metrics.

Event log	Fit(P,L)	Fit2(P,L)	Fit3(P,L)
L1	1.00	1.00	1.00
L2	1.00	1.00	1.00
L3	0.60	0.93	0.93
L4	0.60	0.81	0.87

Table 3.3 : Measuring conformance using three metrics based on underfitting.

By trying to parse cases using the predefined model it is also possible to give diagnostics that may help to locate the source of the misalignment. For this purpose, we use the non-blocking semantics. While parsing the log it is possible to count how many times an activity should have been executed according to the log but could not be executed according to the predefined model. Figure 10 shows that for event log L4 activity I did not fit twice. (Note the “2” next to activity I .) This was caused by the fact that in there was a token missing in place p8. This is indicated by the “-2” next to place p8. For two cases a token was missing in place p8 because activity G did not occur after executing the initial activity A . Therefore, there were two cases where a token was left in place p1 .

This is indicated by the “+2” next to place p1 . Similar diagnostics are shown for event log L4 in Figure 3.7 . Here activity E could not be parsed for the first two cases. In both cases a token was missing in place p4 (see “-2”) and because of the non-blocking semantics two tokens remained in place p6 (see “+2”). Figures 3.7 and 3.8 show that we can pinpoint the parts of the process where there appears to be a misalignment.

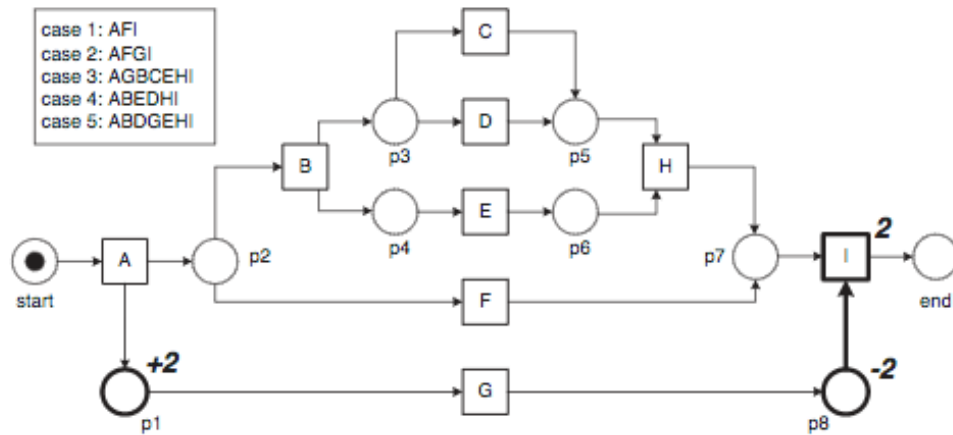


Fig 3.7 Some diagnostics showing the mismatch between Figure 2 and event log L3[4]

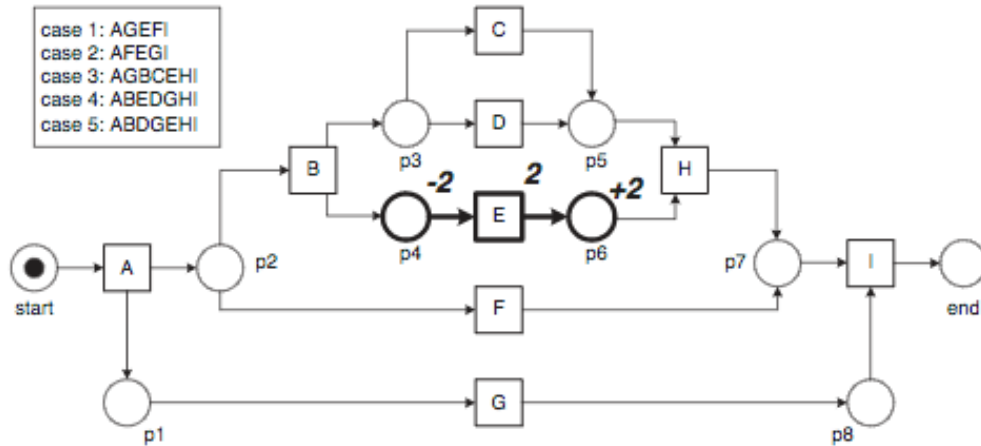


Fig 3.8 : Some diagnostics showing the mismatch between Figure and event log L4 .[3]

In this section, we demonstrated that conformance testing (i.e., comparing the log with some predefined process model) can be used to measure the fitness of the model based on a log. Moreover, the same mechanisms can be used to locate the misalignment.

7. Performance Analysis

State-of-the-art conformance checking techniques create so-called *alignments* between the observed event data and the process model. In an alignment each case is mapped onto the closest path in the model, also in case of deviations and non-determinism. As a result, event logs can always be replayed on process models. This holds for both discovered processes and hand-made models. Next to showing deviations, alignments can also be used to reveal bottlenecks using timestamps of events. Hence, process mining provides novel forms of performance analyses that combine event data with process-centric visualizations

8. Conclusion

Now after the three previous chapters, we have the minimum of requirements for start practice the process mining with real data but the question that assumes is "how we can get the event log and witch a process we have to follow ".we see that in the next chapter .