Distributed Process Discovery and Conformance Checking

Wil M.P. van der Aalst[1,2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
[2] Queensland University of Technology, Brisbane, Australia
www.vdaalst.com

Abstract. Process mining techniques have matured over the last decade and more and more organization started to use this new technology. The two most important types of process mining are *process discovery* (i.e., learning a process model from example behavior recorded in an event log) and *conformance checking* (i.e., comparing modeled behavior with observed behavior). Process mining is motivated by the availability of event data. However, as event logs become larger (say terabytes), performance becomes a concern. The only way to handle larger applications while ensuring acceptable response times, is to *distribute* analysis over a network of computers (e.g., multicore systems, grids, and clouds). This paper provides an overview of the different ways in which process mining problems can be distributed. We identify three types of distribution: *replication*, a *horizontal partitioning* of the event log, and a *vertical partitioning* of the event log. These types are discussed in the context of both *procedural* (e.g., Petri nets) and *declarative* process models. Most challenging is the horizontal partitioning of event logs in the context of procedural models. Therefore, a new approach to decompose Petri nets and associated event logs is presented. This approach illustrates that process mining problems can be distributed in various ways.

Keywords: process mining, distributed computing, grid computing, process discovery, conformance checking, business process management

1 Introduction

Digital data is everywhere – in every sector, in every economy, in every organization, and in every home – and will continue to grow exponentially [22]. Some claim that all of the world's music can be stored on a $600 disk drive. However, despite Moore's Law, storage space and computing power cannot keep up with the growth of event data. Therefore, analysis techniques dealing with "big data" [22] need to resort to distributed computing.

This paper focuses on *process mining*, i.e., the analysis of processes based on *event data* [3]. Process mining techniques aim to *discover, monitor, and improve processes using event logs.* Process mining is a relatively young research discipline that sits between machine learning and data mining on the one hand, and process analysis and formal methods on the other hand. The idea of process mining is to discover, monitor and improve real processes (i.e., not assumed processes) *by extracting knowledge from event logs* readily available in today's (information) systems. Process mining includes (automated) process discovery (i.e., extracting process models from an event log), conformance checking (i.e., monitoring deviations by comparing model and log), social network/organizational mining, automated construction of simulation models, model extension, model repair, case prediction, and history-based recommendations.

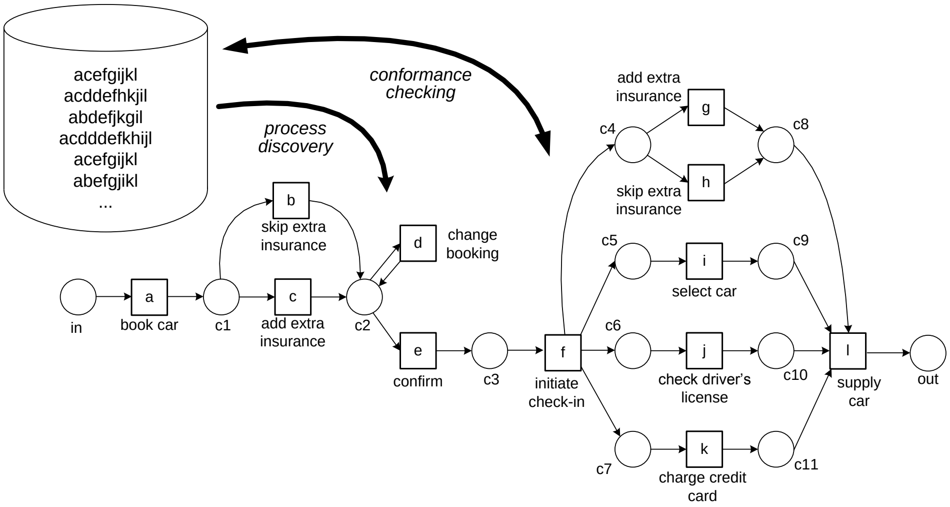


Fig. 1. Example illustrating two types of process mining: process discovery and conformance checking.

Figure 1 illustrates the two most important types of process mining: process discovery and conformance checking. Starting point for process mining is an event log. Each event in such a log refers to an *activity* (i.e., a well-defined step in some process) and is related to a particular *case* (i.e., a *process instance*). The events belonging to a case are *ordered* and can be seen as one "run" of the process. For example, the first case in the event log shown in Fig. 1 can be described by the trace $\langle a, c, e, f, g, i, j, k, l \rangle$. This is the scenario where a car is booked (activity a), extra insurance is added (activity c), the booking is confirmed (activity e), the check-in process is initiated (activity f), more insurance is added (activity g), a car is selected (activity i), the license is checked (activity j), the credit card is charged (activity k), and the car is supplied (activity l). The second case is described by the trace $\langle a, c, d, d, e, f, h, k, j, i, l \rangle$. In this scenario, the booking was changed two times (activity d) and no extra insurance was taken at check-in (activity h). It is important to note that an event log contains only example behavior, i.e., we cannot assume that all possible runs have been observed. In fact, an event log often contains only a fraction of the possible behavior [3].

Process discovery techniques automatically create a model based on the example behavior seen in the event log. For example, based on the event log shown in Fig. 1 the corresponding Petri net is created. Note that the Petri net shown in Fig. 1 is indeed able to generate the behavior in the event log. The model allows for more behavior, but this is often desirable as the model should generalize the observed behavior.

Whereas process discovery constructs a model without any a priori information (other than the event log), conformance checking uses a model and an event log as input. The model may have been made by hand or discovered through process discovery. For conformance checking, the modeled behavior and the observed behavior (i.e., event log) are compared. There are various approaches to diagnose and quantify conformance. For example, one can measure the fraction of cases in the log that can be generated by the model. In Fig. 1, all cases fit the model perfectly. However, if there would have been a case following trace $\langle a, c, f, h, k, j, i, l \rangle$, then conformance checking techniques would identify that in this trace activity e (the confirmation) is missing.

Given a small event log, like the one shown in Fig. 1, analysis is simple. However, in reality, process models may have hundreds of different activities and there may be millions of events and thousands of unique cases. In such cases, process mining techniques may have problems to produce meaningful results in a reasonable time. This is why we are interested in *distributed process mining*, i.e., decomposing challenging process discovery and conformance checking problems into smaller problems that can be distributed over a network of computers.

Today, there are many different types of distributed systems, i.e., systems composed of multiple autonomous computational entities communicating through a network. Multicore computing, cluster computing, grid computing, cloud computing, etc. all refer to systems where different resources are used concurrently to improve performance and scalability. Most data mining techniques can be distributed [16], e.g., there are various techniques for distributed classification, distributed clustering, and distributed association rule mining [13]. However, in the context of process mining only distributed genetic algorithms have been examined in detail [15]. Yet, there is an obvious need for distributed process mining. This paper explores the different ways in which process discovery and conformance checking problems can be distributed. We will not focus on the technical aspects (e.g., the type of distributed system to use) nor on specific mining algorithms. Instead, we systematically explore the different ways in which event logs and models can be partitioned.

The remainder of this paper is organized as follows. First, in Section 2, we discuss the different ways in which process mining techniques can be distributed. Besides *replication*, we define two types of distribution: *vertical distribution* and *horizontal distribution.* In Section 3 we elaborate on the representation of event logs and process models. Here, we also discuss the differences between procedural models and declarative models. We use Petri nets as typical representatives of conventional procedural models. To illustrate the use of declarative models in the context of distributed process mining, we elaborate on the *Declare* language [8]. Section 4 discusses different ways of measuring conformance while zooming in on the notion of fitness. The horizontal distribution of process mining tasks is promising, but also particularly challenging for procedural models. Therefore, we elaborate on a particular technique to decompose event logs and processes (Section 5). Here we use the notion of *passages* for Petri nets which enables us to split event logs and process models horizontally. Section 6 concludes the paper.

2 Distributed Process Mining: An Overview

This section introduces some basic process mining concepts (Section 2.1) and based on these concepts it is shown that event logs and process models can be distributed in various ways (Section 2.2).

2.1 Process Discovery and Conformance Checking

As explained in the introduction there are two basic types of process mining: *process discovery* and *conformance checking*.[3] Figure 2 shows both types.

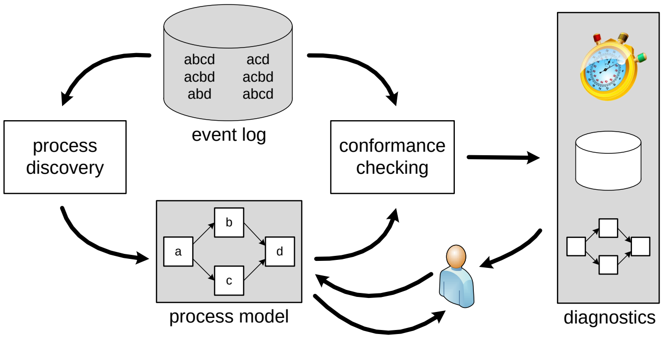


Fig. 2. Positioning process mining techniques.

Process discovery techniques take an event log and produce a process model in some notation. Figure 1 already illustrated the basic idea of discovery: learn a process model from example traces.

Conformance checking techniques take an event log and a process model and compare the observed behavior with the modeled behavior. As Fig. 2 shows

[3] As described in [3], process mining is not limited to process discovery and conformance checking and also includes enhancement (e.g., extending or repairing models based on event data) and operational support (on-the-fly conformance checking, prediction, and recommendation). These are out-of-scope for this paper.

the process model may be the result of process discovery or made by hand. Basically, three types of conformance-related diagnostics can be generated. First of all, there may be overall metrics describing the degree of conformance, e.g., 80% of all cases can be replayed by the model from begin to end. Second, the non-conforming behavior may be highlighted in the event log. Third, the non-conforming behavior may be revealed by annotating the process model. Note that conformance can be viewed from two angles: (a) the model does not capture the real behavior ("the model is wrong") and (b) reality deviates from the desired model ("the event log is wrong"). The first viewpoint is taken when the model is supposed to be descriptive, i.e., capture or predict reality. The second viewpoint is taken when the model is normative, i.e., used to influence or control reality.

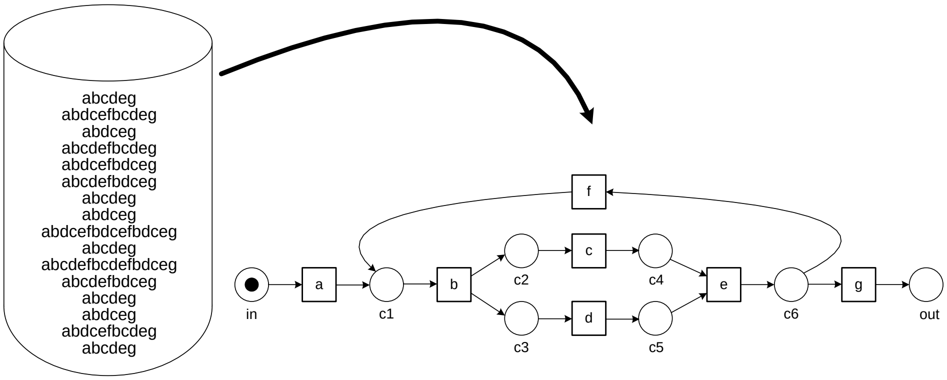


Fig. 3. Example illustrating process discovery.

To further illustrate the notion of process discovery consider the example shown in Fig. 3. Based on the event log shown, a Petri net is learned. Note that all traces in the event log start with activity a and end with activity g. This is also the case in the Petri net (consider all full firing sequences starting with a token in place *in* and ending with a token in *out*). After a, activity b can be executed. Transition b in the Petri net is a so-called AND-split, i.e., after executing b, both c and d can be executed concurrently. Transition e is a so-called AND-join. After executing e a choice is made: either g occurs and the case completes or f is executed and the state with a token in place $c1$ is revisited. Many process discovery algorithms have been proposed in literature [9, 10, 12, 17–19, 23, 28–30]. Most of these algorithms have no problems dealing with this small example.

Figure 4 illustrates conformance checking using the same example. Now the event log contains some traces that are not possible according to the process model shown in Fig. 4. As discussed in the context of Fig. 2, there are three types of diagnostics possible. First of all, we can use metrics to describe the degree of conformance. For example, 10 of the 16 cases (i.e., 62.5 percent) in

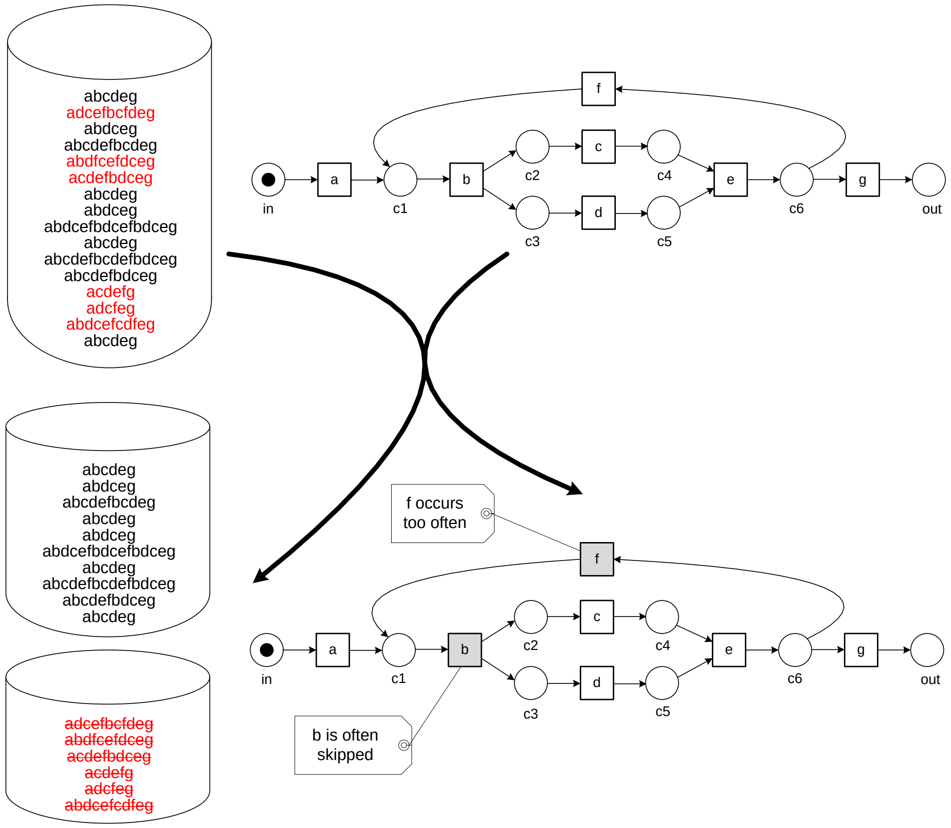


Fig. 4. Example illustrating conformance checking.

Fig. 4 are perfectly fitting. Second, we can split the log into two smaller event logs: one consisting of conforming cases and one consisting of non-conforming cases. These logs can be used for further analysis, e.g., discover commonalities among non-conforming cases using process discovery. Third, we can highlight problems in the model. As Fig. 4 shows, there is a problem with activity b: according to the model b should be executed before c and d but in the event log this is not always the case. There is also a problem with activity f: it should only be executed after e, but in the log it also appears at other places.

Figures 3 and 4 show the basic idea of process mining. Note that the example is oversimplified. For example, most event logs contain much more information. In the example log an event is fully described by an activity name. However, often there is additional information about an event such as the *resource* (i.e., person or device) executing or initiating the activity, the *timestamp* of the event, or *data elements* recorded with the event (e.g., the size of an order).

The process models shown thus far are all Petri nets (WF-nets [1, 6] to be precise). Different process mining algorithms may use different representations.

Moreover, the notation used to visualize the result may be very different from the representation used during the actual discovery process. All mainstream BPM notations (Petri nets, EPCs, BPMN, YAWL, UML activity diagrams, etc.) can be used to show discovered processes [3, 31]. In fact, later we will also elaborate on so-called *declarative* process models. However, to explain the concept of distributed process mining, such differences are less relevant. Therefore, we defer a discussion on the difference between procedural models and declarative models to Section 3.4.

2.2 Distributing Event Logs and Process Models

New computing paradigms such as cloud computing, grid computing, cluster computing, etc. have emerged to perform resource-intensive IT tasks. Modern computers (even lower-end laptops and high-end phones) have multiple processor cores. Therefore, the distribution of computing-intensive tasks, like process mining on "big data", is becoming more important.

At the same time, there is an exponentially growing torrent of event data. MGI estimates that enterprises globally stored more than 7 exabytes of new data on disk drives in 2010, while consumers stored more than 6 exabytes of new data on devices such as PCs and notebooks [22]. A recent study in *Science* suggests that the total global storage capacity increased from 2.6 exabytes in 1986 to 295 exabytes in 2007 [20]. These studies illustrate the growing potential of process mining.

Given these observations, it is interesting to develop techniques for *distributed process mining*. In recent years, distributed data mining techniques have been developed and corresponding infrastructures have been realized [16]. These techniques typically partition the input data over multiple computing nodes. Each of the nodes computes a local model and these local models are aggregated into an overall model.

In [15], we showed that it is fairly easy to distribute genetic process mining algorithms. In this paper (i.e., [15]), we replicate the entire log such that each node has a copy of all input data. Each node runs the same genetic algorithm, uses the whole event log, but, due to randomization, works with different individuals (i.e., process models). Periodically, the best individuals are exchanged between nodes. It is also possible to partition the input data (i.e., the event log) over all nodes. Experimental results show that distributed genetic process mining significantly speeds-up the discovery process. This makes sense because the fitness test is most time-consuming. However, individual fitness tests are completely independent. Although genetic process mining algorithms can be distributed easily, they are not usable for large and complex data sets. Other process mining algorithms tend to outperform genetic algorithms [3]. Therefore, we also need to consider the distribution of other process mining techniques.

To discuss the different ways of distributing process mining techniques we approach the problem from the viewpoint of the event log. We consider three basic types of distribution:

- *Replication.* If the process mining algorithm is non-deterministic, then the same task can be executed on all nodes and in the end the best result can be taken. In this case, the event log can be simply replicated, i.e., all nodes have a copy of the whole event log.
- *Vertical partitioning.* Event logs are composed of cases. There may be thousands or even millions of cases. These can be distributed over the nodes in the network, i.e., each case is assigned to one computing node. All nodes work on a subset of the whole log and in the end the results need to be merged.
- *Horizontal partitioning.* Cases are composed of multiple events. Therefore, we can also partition cases, i.e., part of a case is analyzed on one node whereas another part of the same case is analyzed on another node. In principle, each node needs to consider all cases. However, the attention of one computing node is limited to a particular subset of events per case.

Of course it is possible to combine the three types of distribution.

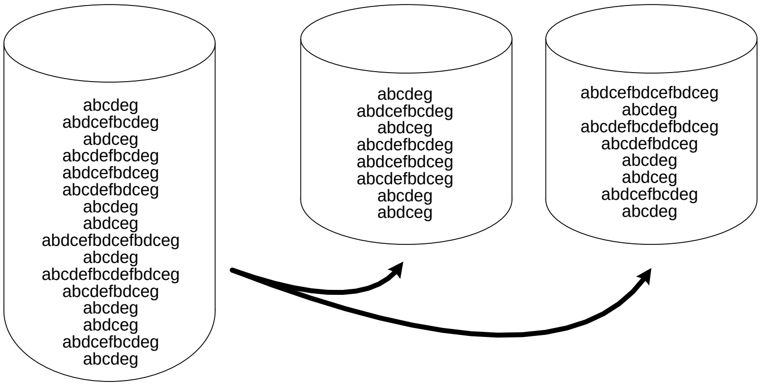


Fig. 5. Partitioning the event log *vertically*: cases are distributed arbitrarily.

Figure 5 illustrates the vertical partitioning of an event log using our running example. The original event log contained 16 cases. Assuming that there are two computing nodes, we can partition the cases over these two nodes. Each case resides in exactly one location, i.e., the nodes operate on disjoint sublogs. Each node computes a process mining result for a sublog and in the end the results are merged. Depending on the type of process mining result, merging may be simple or complex. For example, it we are interested in the percentage of fitting cases it is easy to compute the overall percentage. Suppose there are n nodes and each node $i \in \{1 \ldots n\}$ reports on the number of fitting cases (x_i) and non-fitting cases (y_i) in the sublog. The fraction of fitting cases can be computed easily: $(\sum_i \ x_i)/(\sum_i \ x_i + y_i)$. When each node produces a process model, it is more difficult to produce an overall result. However, by using lower-level output such as the dependency matrices used by mining algorithms like the heuristic miner and fuzzy miner [3], one can merge the results.

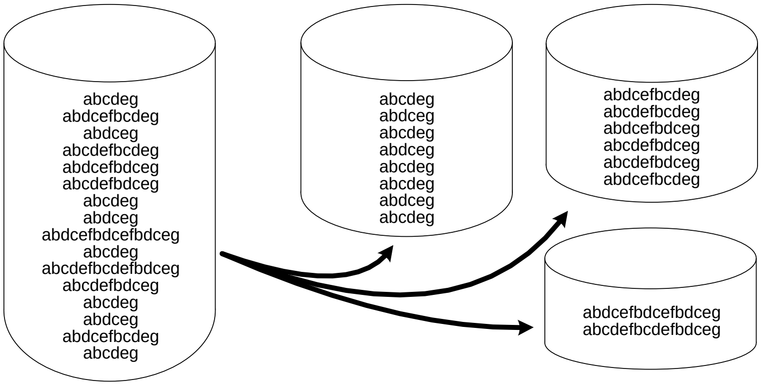


Fig. 6. Partitioning the event log vertically: cases are distributed based on a particular feature (in this case the length of the case).

In Fig. 5 the cases are partitioned over the logs without considering particular features, i.e., the first eight cases are assigned to the first node and the remaining eight cases are assigned to the second node. As Fig. 6 shows, one can also distribute cases based on a particular feature. In this case all cases of length 6 are moved to the first node, cases of length 11 are moved to the second node, and cases of length 16 are moved to the third node. Various features can be used, e.g., the type of customer (one node analyzes the process for gold customers, one for silver customers, etc.), the flow time of the case, the start time of the case, the monetary value of the case, etc. Such a vertical partitioning may provide additional insights. An example is the use of the start time of cases when distributing the event log. Now it is interesting to see whether there are significant differences between the results. The term *concept drift* refers to the situation in which the process is changing while being analyzed [14]. For instance, in the beginning of the event log two activities may be concurrent whereas later in the log these activities become sequential. Processes may change due to periodic/seasonal changes (e.g., “in December there is more demand” or “on Friday afternoon there are fewer employees available”) or due to changing conditions (e.g., “the market is getting more competitive”). A vertical partitioning based on the start time of cases may reveal concept drift or the identification of periods with severe conformance problems.

Figure 7 illustrates the *horizontal* partitioning of event logs. The first sublog contains all events that correspond to activities a, b, e, f, and g. The second sublog contains all events that correspond to activities b, c, d, and e. Note that each case appears in each of the sublogs. However, each sublog contains only a selection of events per case. In other words, events are partitioned “horizontally”

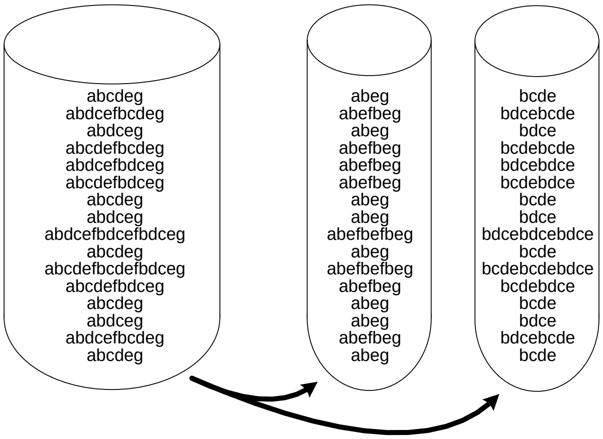


Fig. 7. Partitioning the event log *horizontally*.

instead of “vertically”. Each node computes results for a particular sublog. In the end, all results are merged. Figure 8 shows an example of two process fragments discovered by two different nodes. The process fragments are glued together using the common events. In Section 5 we will further elaborate on this.

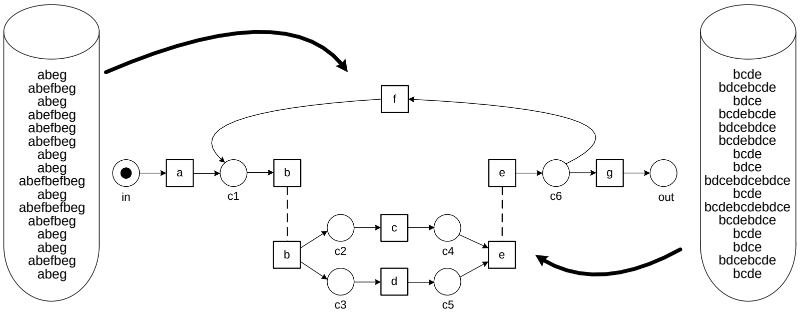


Fig. 8. Horizontally partitioned event logs are used to discover process fragments that can be merged into a complete model.

3 Representation of Event logs and Process Models

Thus far, we have only discussed things informally. In this section, we formalize some of the notions introduced before. For example, we formalize the notion of an event log and provide some Petri net basics. Moreover, we show an example of a declarative language (*Declare* [8]) grounded in LTL.

3.1 Multisets

Multisets are used to represent the state of a Petri net and to describe event logs where the same trace may appear multiple times.

$\mathcal{B}(A)$ is the set of all multisets over some set A. For some multiset $b \in \mathcal{B}(A)$, $b(a)$ denotes the number of times element $a \in A$ appears in b. Some examples: $b_1 = [\ ]$, $b_2 = [x, x, y]$, $b_3 = [x, y, z]$, $b_4 = [x, x, y, x, y, z]$, $b_5 = [x^3, y^2, z]$ are multisets over $A = \{x, y, z\}$. b_1 is the empty multiset, b_2 and b_3 both consist of three elements, and $b_4 = b_5$, i.e., the ordering of elements is irrelevant and a more compact notation may be used for repeating elements.

The standard set operators can be extended to multisets, e.g., $x \in b_2$, $b_2 \uplus b_3 = b_4$, $b_5 \setminus b_2 = b_3$, $|b_5| = 6$, etc. $\{a \in b\}$ denotes the set with all elements a for which $b(a) \geq 1$. $[f(a) \mid a \in b]$ denotes the multiset where element $f(a)$ appears $\sum_{x \in b | f(x)=f(a)} b(x)$ times.

3.2 Event Logs

As indicated earlier, *event logs* serve as the starting point for process mining. An event log is a multiset of *traces*. Each trace describes the life-cycle of a particular *case* (i.e., a *process instance*) in terms of the *activities* executed.

Definition 1 (Trace, Event Log). *Let A be a set of activities. A trace $\sigma \in A^*$ is a sequence of activities. $L \in \mathcal{B}(A^*)$ is an event log, i.e., a multiset of traces.*

An event log is a *multiset* of traces because there can be multiple cases having the same trace. In this simple definition of an event log, an event refers to just an *activity*. Often event logs may store additional information about events. For example, many process mining techniques use extra information such as the *resource* (i.e., person or device) executing or initiating the activity, the *timestamp* of the event, or *data elements* recorded with the event. In this paper, we abstract from such information. However, the results presented in this paper can easily be extended to event logs with more information.

An example log is $L_1 = [\langle a, b, c, d, e, g\rangle^{30}, \langle a, b, d, c, e, g\rangle^{20}, \langle a, b, c, d, e, f, b, c, d, e, g\rangle^{5}, \langle a, b, d, c, e, f, b, c, d, e, g\rangle^{3}, \langle a, b, c, d, e, f, b, d, c, e, g\rangle^{2}]$. L_1 contains information about 60 cases, e.g., 30 cases followed trace $\langle a, b, c, d, e, g\rangle$.

Definition 2 (Projection). *Let A be a set and $X \subseteq A$ a subset. $\upharpoonright_X \in A^* \to X^*$ is a projection function and is defined recursively: (a) $\langle\ \rangle \upharpoonright_X = \langle\ \rangle$ and (b) for $\sigma \in A^*$ and $a \in A$: $(\sigma; \langle a\rangle)\upharpoonright_X = \sigma\upharpoonright_X$ if $a \notin X$ and $(\sigma; \langle a\rangle)\upharpoonright_X = \sigma\upharpoonright_X; \langle a\rangle$ if $a \in X$.*

The projection function is generalized to event logs, i.e., for some event log $L \in \mathcal{B}(A^*)$ and set $X \subseteq A$: $L\upharpoonright_X = [\sigma\upharpoonright_X \mid \sigma \in L]$. For event log L_1 define earlier: $L_1\upharpoonright_{\{a,f,g\}} = [\langle a, g\rangle^{50}, \langle a, f, g\rangle^{10}]$.

3.3 Procedural Models

A wide variety of process modeling languages are used in the context of process mining, e.g., Petri nets, EPCs, C-nets, BPMN, YAWL, and UML activity diagrams [3, 31]. Most of these languages are *procedural* languages (also referred to as *imperative* languages). In this paper, we use Petri nets as a typical representative of such languages. However, the ideas can easily be adapted to fit other languages. Later we will formalize selected distribution concepts in terms of Petri nets. Therefore, we introduce some standard notations.

Definition 3 (Petri Net). *A Petri net is tuple $PN = (P, T, F)$ with P the set of places, T the set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ the flow relation.*

Figure 9 shows an example Petri net. The state of a Petri net, called *marking*, is a multiset of places indicating how many *tokens* each place contains. $[in]$ is the initial marking shown in Fig. 9. Another potential marking is $[c2^{10}, c3^5, c5^5]$. This is the state with ten tokens in $c2$, five tokens in $c3$, and five tokens in $c5$.

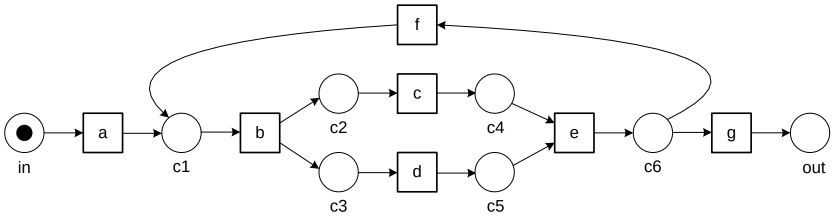


Fig. 9. A Petri net $PN = (P, T, F)$ with $P = \{in, c1, c2, c3, c4, c5, c6, out\}$, $T = \{a, b, c, d, e, f, g\}$, and $F = \{(in, a), (a, c1), (c1, b), \ldots, (g, out)\}$.

Definition 4 (Marking). *Let $PN = (P, T, F)$ be Petri net. A marking M is a multiset of places, i.e., $M \in \mathcal{B}(P)$.*

As usual we define the preset and postset of a node (place or transition) in the Petri net graph. For any $x \in P \cup T$, $\bullet x = \{y \mid (y, x) \in F\}$ (input nodes) and $x\bullet = \{y \mid (x, y) \in F\}$ (output nodes).

A transition $t \in T$ is *enabled* in marking M, denoted as $M[t\rangle$, if each of its input places $\bullet t$ contains at least one token. Consider the Petri net in Fig. 9 with $M = [c3, c4]$: $M[e\rangle$ because *both* input places are marked.

An enabled transition t may *fire*, i.e., one token is removed from each of the input places $\bullet t$ and one token is produced for each of the output places $t\bullet$. Formally: $M' = (M \setminus \bullet t) \uplus t\bullet$ is the marking resulting from firing enabled transition t in marking M. $M[t\rangle M'$ denotes that t is enabled in M and firing t results in marking M'. For example, $[in][a\rangle[c1]$ and $[c1][b\rangle[c2, c3]$ for the net in Fig. 9.

Let $\sigma = \langle t_1, t_2, \ldots, t_n \rangle \in T^*$ be a sequence of transitions. $M[\sigma\rangle M'$ denotes that there is a set of markings $M_0, M_1, \ldots, M_n$ such that $M_0 = M$, $M_n = M'$, and $M_i[t_{i+1}\rangle M_{i+1}$ for $0 \leq i < n$. A marking M' is *reachable* from M if there exists a σ such that $M[\sigma\rangle M'$. For example, $[in][\sigma\rangle[out]$ for $\sigma = \langle a, b, c, d, e, g \rangle$.

Definition 5 (Labeled Petri Net). *A labeled Petri net $PN = (P, T, F, T_v)$ is a Petri net (P, T, F) with visible labels $T_v \subseteq T$. Let $\sigma_v = \langle t_1, t_2, \ldots, t_n \rangle \in T_v^*$ be a sequence of visible transitions. $M[\sigma_v \rhd M'$ if and only if there is a sequence $\sigma \in T^*$ such that $M[\sigma\rangle M'$ and the projection of σ on T_v yields σ_v (i.e., $\sigma_v = \sigma\restriction_{T_v}$).*

If we assume $T_v = \{a, e, f, g\}$ for the Petri net in Fig. 9, then $[in][\sigma_v \rhd [out]$ for $\sigma_v = \langle a, e, f, e, f, e, g \rangle$ (i.e., b, c, and d are invisible).

In the context of process mining, we always consider processes that start in an initial state and end in a well-defined end state. For example, given the net in Fig. 9 we are interested in firing sequences starting in $M_i = [in]$ and ending in $M_o = [out]$. Therefore, we define the notion of a *system net*.

Definition 6 (System Net). *A system net is a triplet $SN = (PN, M_i, M_o)$ where $PN = (P, T, F, T_v)$ is a Petri net with visible labels T_v, $M_i \in \mathcal{B}(P)$ is the initial marking, and $M_o \in \mathcal{B}(P)$ is the final marking.*

Given a system net, $\tau(SN)$ is the set of all possible visible full traces, i.e., firing sequences starting in M_i and ending in M_o projected onto the set of visible transitions.

Definition 7 (Traces). *Let $SN = (PN, M_i, M_o)$ be a system net. $\tau(SN) = \{\sigma_v \mid M_i[\sigma_v \rhd M_o\}$ is the set of visible traces starting in M_i and ending in M_o.*

If we assume $T_v = \{a, e, g\}$ for the Petri net in Fig. 9, then $\tau(SN) = \{\langle a, e, g \rangle, \langle a, e, e, g \rangle, \langle a, e, e, e, g \rangle, \ldots\}$.

The Petri net in Fig. 9 has a designated source place (*in*), a designated source place (*out*), and all nodes are on a path from *in* to *out*. Such nets are called *WF-nets* [1, 6].

Definition 8 (WF-net). *$WF = (PN, in, T_i, out, T_o)$ is a workflow net (WF-net) if*

- *$PN = (P, T, F, T_v)$ is a labeled Petri net,*
- *$in \in P$ is a source place such that $\bullet in = \emptyset$ and $in\bullet = T_i$,*
- *$out \in P$ is a sink place such that $out\bullet = \emptyset$ and $\bullet out = T_o$,*
- *$T_i \subseteq T_v$ is the set of initial transitions and $\bullet T_i = \{in\}$,*
- *$T_o \subseteq T_v$ is the set of final transitions and $T_o\bullet = \{out\}$, and*
- *all nodes are on some path from source place in to sink place out.*

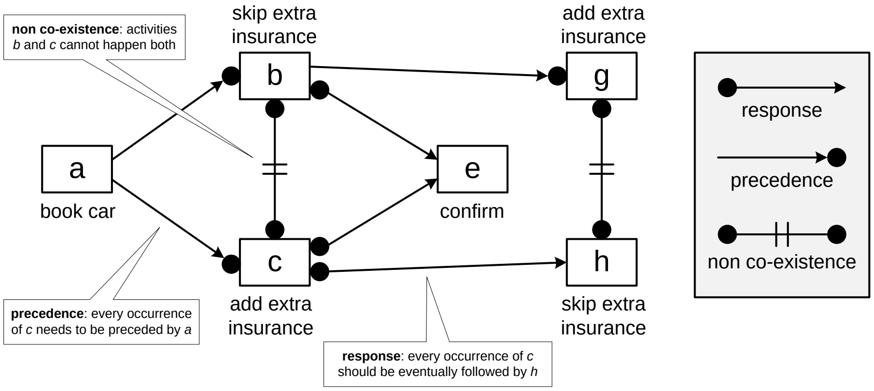
WF-nets are often used in the context of business process modeling and process mining. Compared to the standard definition of WF-nets [1, 6] we added the requirement that the initial and final transitions need to be visible.

A WF-net $WF = (PN, in, T_i, out, T_o)$ defines the system $SN = (PN, M_i, M_o)$ with $M_i = [in]$ and $M_o = [out]$. Ideally WF-nets are also *sound*, i.e., free of deadlocks, livelocks, and other anomalies [1, 6]. Formally, this means that it is possible to reach M_o from any state reachable from M_i.

Process models discovered using existing process mining techniques may be unsound. Therefore, we cannot assume/require all WF-nets to be sound.

3.4 Declarative Models

Procedural process models (like Petri nets) take an "inside-to-outside" approach, i.e., all execution alternatives need to be specified explicitly and new alternatives must be explicitly added to the model. Declarative models use an "outside-to-inside" approach: anything is possible unless explicitly forbidden. Declarative models are particularly useful for conformance checking. Therefore, we elaborate on *Declare*. Declare is both a language (in fact a family of languages) and a fully functional WFM system [8, 24].



non co-existence: activities *b* and *c* cannot happen both

precedence: every occurrence of *c* needs to be preceded by *a*

response: every occurrence of *c* should be eventually followed by *h*

Fig. 10. Example of a Declare model consisting of six activities and eight constraints.

Declare uses a graphical notation and its semantics are based on LTL (Linear Temporal Logic) [8]. Figure 10 shows a Declare specification consisting of eight constraints. The construct connecting activities b and c is a so-called *non-coexistence constraint*. In terms of LTL this constraint means "$\neg((\Diamond b) \wedge (\Diamond c))$"; $\Diamond b$ and $\Diamond c$ cannot both be true, i.e., it cannot be the case that both b and c happen for the same case. There is also a non-coexistence constraint preventing the execution of both g and h for the same case. There are three *precedence constraints*. The semantics of the precedence constraint connecting a to b can also be expressed in terms of LTL: "$(\neg b)\ W\ a$", i.e., b should not happen before a has happened. Since the weak until (W) is used in "$(\neg b)\ W\ a$", traces without any a and b events also satisfy the constraint. Similarly, g should not happen before b has happened: "$(\neg g)\ W\ b$". There are three *response constraints*. The LTL formalization of the precedence constraint connecting b to e is "$\Box(b \Rightarrow (\Diamond e))$", i.e., every occurrence of b should eventually be followed by e. Note that the behavior generated by the WF-net in Fig. 1 satisfies all constraints specified in the Declare model, i.e., none of the eight constraints is violated by any of the traces. However, the Declare model shown in Figure 10 allows for all kinds of behaviors not possible in Fig. 1. For example, trace $\langle a, a, b, e, e, g, g\rangle$ is allowed. Whereas in a procedural model, everything is forbidden unless explicitly enabled, a declarative model allows for anything unless explicitly forbidden. For processes with a lot of flexibility, declarative models are more appropriate [8, 24].

In [5] it is described how Declare/LTL constraints can be checked for a given log. This can also be extended to the on-the-fly conformance checking. Consider some running case having a partial trace $\sigma_p \in A^*$ listing the events that have happened thus far. Each constraint c is in one of the following states for σ_p:

- *Satisfied*: the LTL formula corresponding to c evaluates to true for the partial trace σ_p.
- *Temporarily violated*: the LTL formula corresponding to c evaluates to false for σ_p, however, there is a longer trace σ'_p that has σ_p as a prefix and for which the LTL formula corresponding to c evaluates to true.
- *Permanently violated*: the LTL formula corresponding to c evaluates to false for σ_p and all its extensions, i.e., there is no σ'_p that has σ_p as a prefix and for which the LTL formula evaluates to true.

These three notions can be lifted from the level of a *single constraint* to the level of a *complete Declare specification*, e.g., a Declare specification is *satisfied* for a case if all of its constraints are satisfied. This way it is possible to check conformance on-the-fly and generate warnings the moment constraints are permanently/temporarily violated [3].

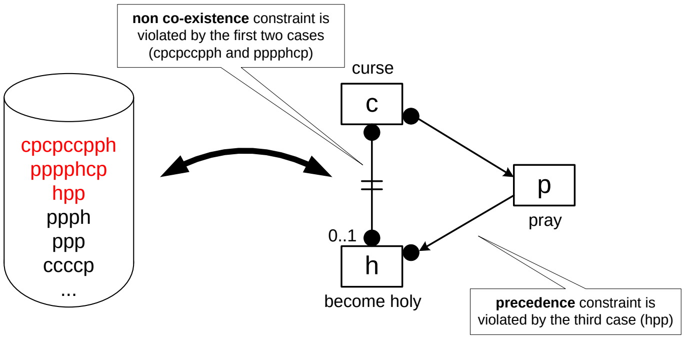

non co-existence constraint is violated by the first two cases (cpcpccpph and pppphcp)
curse
c
p
pray
0..1
h
become holy
precedence constraint is violated by the third case (hpp)
cpcpccpph
pppphcp
hpp
ppph
ppp
ccccp
...

Fig. 11. Conformance checking using a declarative model.

We use the smaller example shown in Fig. 11 to illustrate conformance checking in the context of Declare. The process model shows four constraints: the same person cannot “curse” and “become holy” (non-coexistence constraint), after one “curses” one should eventually “pray” (response constraint), one can only "become holy" after having "prayed" at least once (precedence constraint), and activity h ("become holy") can be executed at most once (cardinality constraint).

Two of the four constraints are violated by the event log shown in Fig. 11. The first two traces/persons cursed and became holy at the same time. The third trace/person became holy without having prayed before.

Conformance checking can be distributed easily for declarative models. One can partition the log *vertically* and simply check per computing node all constraints on the corresponding sublog. One can also partition the set of constraints. Each node of the computer network is responsible for a subset of the constraints and uses a log projected onto the relevant activities, i.e., the event log is distributed *horizontally*. In both cases, it is easy to aggregate the results into overall diagnostics.

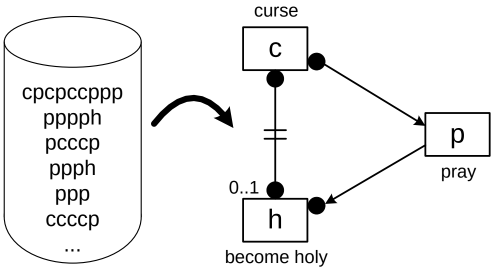


Fig. 12. Discovering a declarative model.

Figure 12 illustrates the discovery of Declare constraints from event logs [21]. A primitive discovery approach is to simply investigate a large collection of candidate constraints using conformance checking. This can be distributed vertically or horizontally as just described. It is also possible to use smarter approaches using the "interestingness" of potential constraints. Here ideas from distributed association rule mining [13] can be employed.

4 Measuring Conformance

Conformance checking techniques can be used to investigate how well an event log $L \in \mathcal{B}(A^*)$ and the behavior allowed by a model fit together. Figure 4 shows an example where deviations between an event log and Petri net are diagnosed. Figure 11 shows a similar example but now using a Declare model. Both examples focus on a particular conformance notion: *fitness*. A model with good fitness allows for most of the behavior seen in the event log. A model has a *perfect fitness* if all traces in the log can be replayed by the model from beginning to end. This notion can be formalized as follows.

Definition 9 (Perfectly Fitting Log). *Let $L \in \mathcal{B}(A^*)$ be an event log and let $SN = (PN, M_i, M_o)$ be a system net. L is perfectly fitting SN if and only if $\{\sigma \in L\} \subseteq \tau(SN)$.*

The above definition assumes a Petri net as process model. However, the same idea can be operationalized for Declare models [5], i.e., for each constraint and every case the corresponding LTL formula should hold.

Consider two event logs $L_1 = [\langle a,c,d,g\rangle^{30}, \langle a,d,c,g\rangle^{20}, \langle a,c,d,f,c,d,g\rangle^{5}, \langle a,d,c,f,c,d,g\rangle^{3}, \langle a,c,d,f,d,c,g\rangle^{2}]$ and $L_2 = [\langle a,c,d,g\rangle^{8}, \langle a,c,g\rangle^{6}, \langle a,c,f,d,g\rangle^{5}]$ and the system net SN of the WF-net depicted in Fig. 9 with $T_v = \{a,c,d,f,g\}$. Clearly, L_1 is perfectly fitting SN whereas L_2 is not. There are various ways to quantify fitness [3, 4, 11, 19, 23, 25–27], typically on a scale from 0 to 1 where 1 means perfect fitness. To measure fitness, one needs to *align* traces in the event log to traces of the process model. Some example alignments for L_2 and SN:

$$\gamma_1 = \begin{array}{|c|c|c|c|} a & c & d & g \\ \hline a & c & d & g \end{array} \quad \gamma_2 = \begin{array}{|c|c|c|c|} a & c & \gg & g \\ \hline a & c & d & g \end{array} \quad \gamma_3 = \begin{array}{|c|c|c|c|c|} a & c & f & d & g \\ \hline a & c & \gg & d & g \end{array} \quad \gamma_4 = \begin{array}{|c|c|c|c|c|c|c|} a & c & \gg & f & d & \gg & g \\ \hline a & c & d & f & d & c & g \end{array}$$

The top row of each alignment corresponds to "moves in the log" and the bottom row corresponds to "moves in the model". If a move in the log cannot be mimicked by a move in the model, then a "$\gg$" ("no move") appears in the bottom row. For example, in γ_3 the model is unable to do f in-between c and d. If a move in the model cannot be mimicked by a move in the log, then a "$\gg$" ("no move") appears in the top row. For example, in γ_2 the log did not do a d move whereas the model has to make this move to enable g and reach the end. Given a trace in the event log, there may be many possible alignments. The goal is to find the alignment with the least number of $\gg$ elements, e.g., γ_3 seems better than γ_4. Finding a optimal alignment can be viewed as an optimization problem [4, 11]. After selecting an optimal alignment, the number of $\gg$ elements can be used to quantify fitness.

Fitness is just one of the four basic conformance dimensions defined in [3]. Other quality dimensions for comparing model and log are *simplicity*, *precision*, and *generalization*.

The *simplest* model that can explain the behavior seen in the log is the best model. This principle is known as Occam's Razor. There are various metrics to quantify the complexity of a model (e.g., size, density, etc.).

The *precision* dimension is related to the desire to avoid "underfitting". It is very easy to construct an extremely simple Petri net ("flower model") that is able to replay all traces in an event log (but also any other event log referring to the same set of activities). See [4, 25–27] for metrics quantifying this dimension.

The *generalization* dimension is related to the desire to avoid "overfitting". In general it is undesirable to have a model that only allows for the exact behavior seen in the event log. Remember that the log contains only example behavior and that many traces that are possible may not have been seen yet.

Conformance checking can be done for various reasons. First of all, it may be used to audit processes to see whether reality conforms to some normative of descriptive model [7]. Deviations may point to fraud, inefficiencies, and poorly designed or outdated procedures. Second, conformance checking can be used to evaluate the performance of a process discovery technique. In fact, genetic process mining algorithms use conformance checking to select the candidate models used to create the next generation of models [23].

5 Example: Horizontal Distribution Using Passages

The vertical distribution of process mining tasks is often fairly straightforward; just partition the event log and run the usual algorithms on each sublog residing at a particular node in the computer network. The horizontal partitioning of event logs is more challenging, but potentially very attractive as the focus of analysis can be limited to a few activities per node. Therefore, we describe a generic distribution approach based on the notion of *passages*.

5.1 Passages in Graphs

A *graph* is a pair $G = (N, E)$ comprising a set N of *nodes* and a set $E \subseteq N \times N$ of *edges*. A Petri net (P, T, F) can be seen as a particular graph with nodes $N = P \cup T$ and edges $E = F$. Like for Petri nets, we define preset $\bullet n = \{n' \in N \mid (n', n) \in E\}$ (direct predecessors) and postset $n\bullet = \{n' \in N \mid (n, n') \in E\}$ (direct successors). This can be generalized to sets, i.e., for $X \subseteq N$: $\bullet X = \cup_{n \in X} \bullet n$ and $X\bullet = \cup_{n \in X} n\bullet$.

To decompose process mining problems into smaller problems, we partition process models using the notion *passages*. A passage is a pair of non-empty sets of nodes (X, Y) such that the set of direct successors of X is Y and the set of direct predecessors of Y is X.

Definition 10 (Passage). *Let $G = (N, E)$ be a graph. $P = (X, Y)$ is a passage if and only if $\emptyset \neq X \subseteq N$, $\emptyset \neq Y \subseteq N$, $X\bullet = Y$, and $X = \bullet Y$. $pas(G)$ is the set of all passages of G.*

Consider the sets $X = \{a, b, c, e, f, g\}$ and $Y = \{c, d, g, h, i\}$ in the graph fragment shown in Fig. 13. (X, Y) is a passage. As indicated, there may be no edges leaving from X to nodes outside Y and there may be no edges into Y from nodes outside X.

Definition 11 (Operations on Passages). *Let $P_1 = (X_1, Y_1)$ and $P_2 = (X_2, Y_2)$ be two passages.*

- *$P_1 \leq P_2$ if and only if $X_1 \subseteq X_2$ and $Y_1 \subseteq Y_2$,*
- *$P_1 < P_2$ if and only if $P_1 \leq P_2$ and $P_1 \neq P_2$,*
- *$P_1 \cup P_2 = (X_1 \cup X_2, Y_1 \cup Y_2)$,*
- *$P_1 \setminus P_2 = (X_1 \setminus X_2, Y_1 \setminus Y_2)$.*

The union of two passages $P_1 \cup P_2$ is again a passage. The difference of two passages $P_1 \setminus P_2$ is a passage if $P_2 < P_1$. Since the union of two passages is again a passage, it is interesting to consider *minimal passages*. A passage is *minimal* if it does not “contain” a smaller passage.

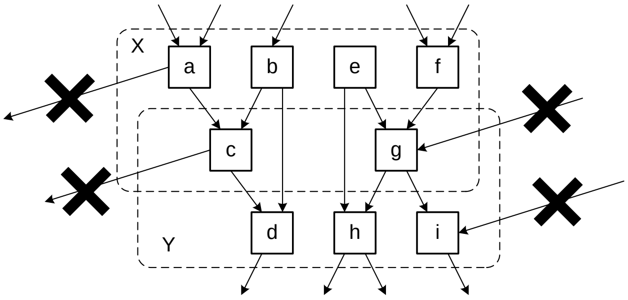


Fig. 13. (X, Y) is a passage because $X\bullet = \{a, b, c, e, f, g\}\bullet = \{c, d, g, h, i\} = Y$ and $X = \{a, b, c, e, f, g\} = \bullet\{c, d, g, h, i\} = \bullet Y$.

Definition 12 (Minimal Passage). *Let $G = (N, E)$ be a graph with passages $pas(G)$. $P \in pas(G)$ is* minimal *if there is no $P' \in pas(G)$ such that $P' < P$. $pas_{min}(G)$ is the set of minimal passages.*

The passage in Figure 13 is not minimal. It can be split into the passages $(\{a, b, c\}, \{c, d\})$ and $(\{e, f, g\}, \{g, h, i\})$. An edge uniquely determines one minimal passage.

Lemma 1. *Let $G = (N, E)$ be a graph and $(x, y) \in E$. There is precisely one minimal passage $P_{(x,y)} = (X, Y) \in pas_{min}(G)$ such that $x \in X$ and $y \in Y$.*

Passages define an equivalence relation on the edges in a graph: $(x_1, y_1) \sim (x_2, y_2)$ if and only if $P_{(x_1,y_1)} = P_{(x_2,y_2)}$. For any $\{(x, y), (x', y), (x, y')\} \subseteq E$: $P_{(x,y)} = P_{(x',y)} = P_{(x,y')}$, i.e., $P_{(x,y)}$ is uniquely determined by x and $P_{(x,y)}$ is also uniquely determined by y. Moreover, $pas_{min}(G) = \{P_{(x,y)} \mid (x, y) \in E\}$.

5.2 Distributed Conformance Checking Using Passages

Now we show that it is possible to decompose and distribute conformance checking problems using the notion of *passages*. In order to do this we focus on the visible transitions and create the so-called *skeleton* of the process model. To define skeletons, we introduce the notation $x \overset{\sigma:E\#Q}{\rightsquigarrow} y$ which states that there is a non-empty path σ from node x to node y where the set of intermediate nodes visited by path σ does not include any nodes in Q.

Definition 13 (Path). *Let $G = (N, E)$ be a graph with $x, y \in N$ and $Q \subseteq N$. $x \overset{\sigma:E\#Q}{\rightsquigarrow} y$ if and only if there is a sequence $\sigma = \langle n_1, n_2, \dots n_k\rangle$ with $k > 1$ such that $x = n_1$, $y = n_k$, for all $1 \leq i < k$: $(n_i, n_{i+1}) \in E$, and for all $1 < i < k$: $n_i \notin Q$. Derived notations:*

- *$x \overset{E\#Q}{\rightsquigarrow} y$ if and only if there exists a path σ such that $x \overset{\sigma:E\#Q}{\rightsquigarrow} y$,*
- *$nodes(x \overset{E\#Q}{\rightsquigarrow} y) = \{n \in \sigma \mid \exists_{\sigma \in N^*}\ x \overset{\sigma:E\#Q}{\rightsquigarrow} y\}$, and*

– *for* $X, Y \subseteq N$: $nodes(X \overset{E\#Q}{\rightsquigarrow} Y) = \cup_{(x,y)\in X\times Y}\ nodes(x \overset{E\#Q}{\rightsquigarrow} y)$.

Definition 14 (Skeleton). *Let* $PN = (P, T, F, T_v)$ *be a labeled Petri net. The* skeleton *of* PN *is the graph* $skel(PN) = (N, E)$ *with* $N = T_v$ *and* $E = \{(x, y) \in T_v \times T_v \mid x \overset{F\#T_v}{\rightsquigarrow} y\}$.

Figure 14 shows the skeleton of the WF-net in Fig. 1 assuming that $T_v = \{a, b, c, d, e, f, l\}$. The resulting graph has four minimal passages.

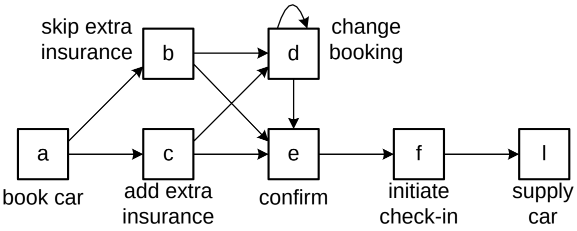


Fig. 14. The skeleton of the labeled Petri net in Fig. 1 (assuming that $T_v = \{a, b, c, d, e, f, l\}$). There are four minimal passages: $(\{a\}, \{b, c\})$, $(\{b, c, d\}, \{d, e\})$, $(\{e\}, \{f\})$, and $(\{f\}, \{l\})$.

Note that only the visible transitions T_v appear in the skeleton. For example, if we assume that $T_v = \{a, f, l\}$ in Fig. 1, then the skeleton is $(\{a, f, l\}, \{(a, f), (f, l)\})$ with only two passages $(\{a\}, \{f\})$ and $(\{f\}, \{l\})$.

If there are multiple minimal passages in the skeleton, we can decompose conformance checking problems into smaller problems *by partitioning the Petri net into net fragments and the event log into sublogs*. Each passage (X, Y) defines one net fragment $PN^{(X,Y)}$ and one sublog $L\restriction_{X\cup Y}$. We will show that conformance can be checked per passage.

Consider event log $L = [\langle a, b, e, f, l\rangle^{20}, \langle a, c, e, f, l\rangle^{15}, \langle a, b, d, e, f, l\rangle^{5}, \langle a, c, d, e, f, l\rangle^{3}, \langle a, b, d, d, e, f, l\rangle^{2}]$, the WF-net PN shown in Fig. 1 with $T_v = \{a, b, c, d, e, f, l\}$, and the skeleton shown in Fig. 14. Based on the four passages, we define four net fragments PN_1, PN_2, PN_3 and PN_4 as shown in Fig. 15. Moreover, we define four sublogs: $L_1 = [\langle a, b\rangle^{27}, \langle a, c\rangle^{18}]$, $L_2 = [\langle b, e\rangle^{20}, \langle c, e\rangle^{15}, \langle b, d, e\rangle^{5}, \langle c, d, e\rangle^{3}, \langle b, d, d, e\rangle^{2}]$, $L_3 = [\langle e, f\rangle^{45}]$, and $L_4 = [\langle f, l\rangle^{45}]$. To check the conformance of the overall event log on the overall model, we check the conformance of L_i on PN_i for $i \in \{1, 2, 3, 4\}$. Since L_i is perfectly fitting PN_i for all i, we can conclude that L is perfectly fitting PN. This illustrates that conformance checking can indeed be decomposed. To formalize this result, we define the notion of a net fragment corresponding to a passage.

Definition 15 (Net Fragment). *Let* $PN = (P, T, F, T_v)$ *be a labeled Petri net. For any two sets of transitions* $X, Y \subseteq T_v$, *we define the net fragment* $PN^{(X,Y)} = (P', T', F', T'_v)$ *with:*

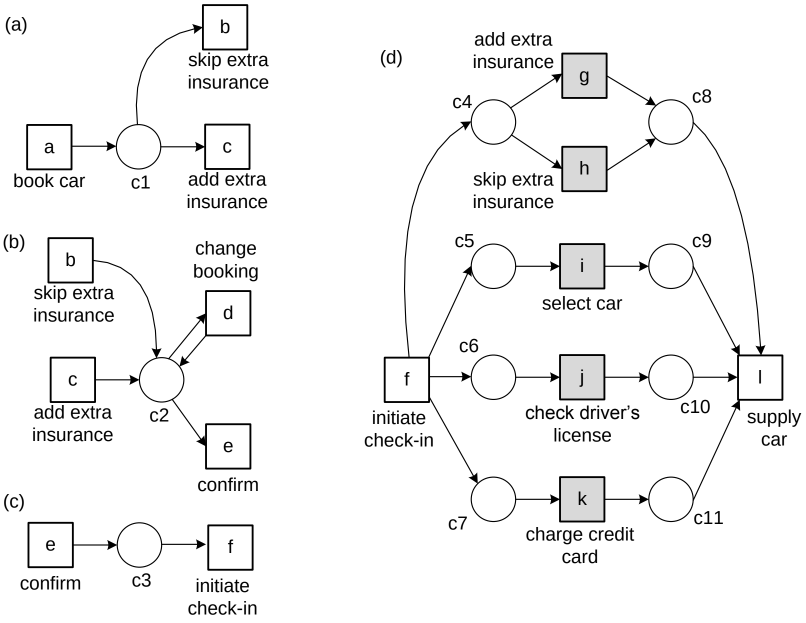


Fig. 15. Four net fragments corresponding to the four passages of the skeleton in Fig. 14: (a) $PN_1 = PN^{(\{a\},\{b,c\})}$, (b) $PN_2 = PN^{(\{b,c,d\},\{d,e\})}$, (c) $PN_3 = PN^{(\{e\},\{f\})}$, and (c) $PN_4 = PN^{(\{f\},\{l\})}$. The invisible transitions, i.e., the transitions in $T \setminus T_v$, are shaded.

- $Z = nodes(X \overset{F\#T_v}{\rightsquigarrow} Y) \setminus (X \cup Y)$ *are the internal nodes of the fragment,*
- $P' = P \cap Z$,
- $T' = (T \cap Z) \cup X \cup Y$,
- $F' = F \cap ((P' \times T') \cup (T' \times P'))$, *and*
- $T'_v = X \cup Y$.

A process model can be decomposed into net fragments corresponding to minimal passages and an event log can be decomposed by projecting the traces on the activities in these minimal passages. The following theorem shows that conformance checking can be done per passage.

Theorem 1 (Distributed Conformance Checking). *Let $L \in \mathcal{B}(A^*)$ be an event log and let $WF = (PN, in, T_i, out, T_o)$ be a WF-net with $PN = (P, T, F, T_v)$. L is perfectly fitting system net $SN = (PN, [in], [out])$ if and only if*

- *for any $\langle a_1, a_2, \ldots a_k \rangle \in L$: $a_1 \in T_i$ and $a_k \in T_o$, and*
- *for any $(X, Y) \in pas_{min}(skel(PN))$: $L\restriction_{X \cup Y}$ is perfectly fitting $SN^{(X,Y)} = (PN^{(X,Y)}, [\,], [\,])$.*

For a formal proof, we refer to [2]. Although the theorem only addresses the notion of perfect fitness, other conformance notions can be decomposed in a similar manner. Metrics can be computed per passage and then aggregated into an overall metric.

Assuming a process model with many passages, the time needed for conformance checking can be reduced significantly. There are two reasons for this. First of all, as Theorem 1 shows, larger problems can be decomposed into a set of independent smaller problems. Therefore, conformance checking can be distributed over multiple computers. Second, due to the exponential nature of most conformance checking techniques, the time needed to solve "many smaller problems" is less than the time needed to solve "one big problem". Existing approaches use state-space analysis (e.g., in [27] the shortest path enabling a transition is computed) or optimization over all possible alignments (e.g., in [11] the A^* algorithm is used to find the best alignment). These techniques do *not* scale linearly in the number of activities. *Therefore, decomposition is useful even if the checks per passage are done on a single computer.*

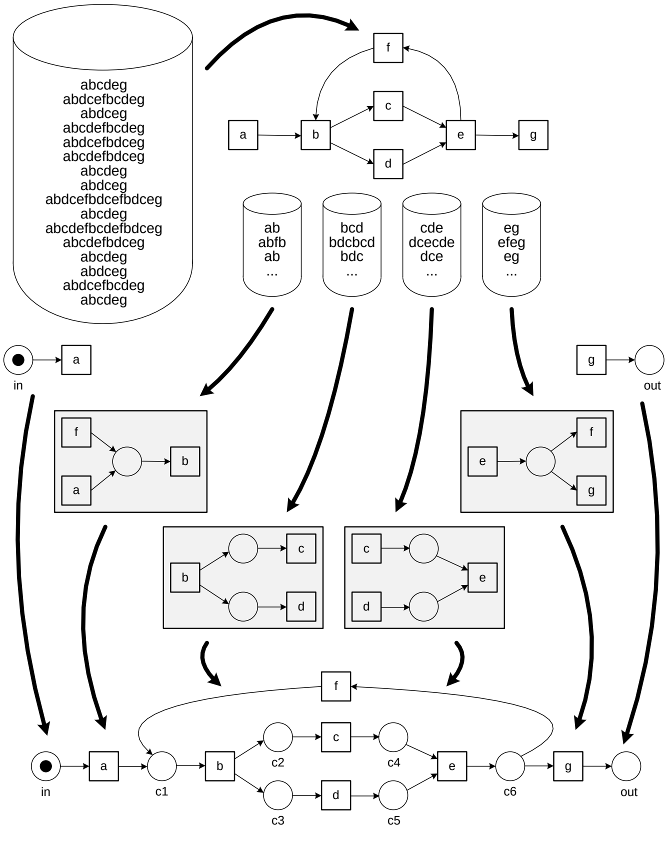


Fig. 16. Distributed discovery based on four minimal passages: $(\{a, f\}, \{b\})$, $(\{b\}, \{c, d\})$, $(\{c, d\}, \{e\})$, and $(\{e\}, \{f, g\})$. A process fragment is discovered for each passage. Subsequently, the fragments are merged into one overall process.

5.3 Distributed Process Discovery Using Passages

As explained before, conformance checking and process discovery are closely related. Therefore, we can exploit the approach used in Theorem 1 for process discovery provided that some coarse *causal structure* (comparable to the skeleton in Section 5.2) is known. There are various techniques to extract such a causal structure, see for example the dependency relations used by the heuristic miner [29]. The causal structure defines a collection of passages and the detailed discovery can be done per passage. Hence, the discovery process can be distributed. The idea is illustrated in Fig. 16.

The approach is independent of the discovery algorithm used. The only assumption is that the casual structure can be determined upfront. See [2] for more details.

By decomposing the overall discovery problem into a collection of smaller discovery problems, it is possible to do a more refined analysis and achieve significant speed-ups. The discovery algorithm is applied to an event log consisting of just the activities involved in the passage under investigation. Hence, process discovery tasks can be distributed over a network of computers (assuming there are multiple passages). Moreover, most discovery algorithms are exponential in the number of activities. Therefore, the sequential discovery of all individual passages is still faster than solving one big discovery problem.

6 Conclusion

This paper provides an overview of the different mechanisms to distribute process mining tasks over a set of computing nodes. Event logs can be decomposed vertically and horizontally. In a vertically distributed event log, each case is analyzed by a designated computing node in the network and each node considers the whole process model (all activities). In a horizontally distributed event log, the cases themselves are partitioned and each node considers only a part of the overall process model. These distribution approaches are fairly independent of the mining algorithm and apply to both procedural and declarative languages. Most challenging is the horizontal distribution of event logs while using a procedural language. However, as shown in this paper, it is still possible to horizontally distribute process discovery and conformance checking tasks using the notion of passages.

Acknowledgments. The author would like to thank all that contributed to the ProM toolset. Many of their contributions are referred to in this paper. Special thanks go to Boudewijn van Dongen and Eric Verbeek (for their work on the ProM infrastructure), Carmen Bratosin (for her work on distributed genetic mining), Arya Adriansyah and Anne Rozinat (for their work on conformance checking), and Maja Pesic, Fabrizio Maggi, and Michael Westergaard (for their work on Declare).

References

1. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
2. W.M.P. van der Aalst. Decomposing Process Mining Problems Using Passages. BPM Center Report BPM-11-19, BPMcenter.org, 2011.
3. W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer-Verlag, Berlin, 2011.
4. W.M.P. van der Aalst, A. Adriansyah, and B. van Dongen. Replaying History on Process Models for Conformance Checking and Performance Analysis. *WIREs Data Mining and Knowledge Discovery*, 2012.
5. W.M.P. van der Aalst, H.T. de Beer, and B.F. van Dongen. Process Mining and Verification of Properties: An Approach based on Temporal Logic. In R. Meersman and Z. Tari et al., editors, *On the Move to Meaningful Internet Systems (CoopIS 2005)*, volume 3760 of *Lecture Notes in Computer Science*, pages 130–147. Springer-Verlag, Berlin, 2005.
6. W.M.P. van der Aalst, K.M. van Hee, A.H.M. ter Hofstede, N. Sidorova, H.M.W. Verbeek, M. Voorhoeve, and M.T. Wynn. Soundness of Workflow Nets: Classification, Decidability, and Analysis. *Formal Aspects of Computing*, 23(3):333–363, 2011.
7. W.M.P. van der Aalst, K.M. van Hee, J.M. van der Werf, and M. Verdonk. Auditing 2.0: Using Process Mining to Support Tomorrow's Auditor. *IEEE Computer*, 43(3):90–93, 2010.
8. W.M.P. van der Aalst, M. Pesic, and H. Schonenberg. Declarative Workflows: Balancing Between Flexibility and Support. *Computer Science - Research and Development*, 23(2):99–113, 2009.
9. W.M.P. van der Aalst, V. Rubin, H.M.W. Verbeek, B.F. van Dongen, E. Kindler, and C.W. Günther. Process Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting. *Software and Systems Modeling*, 9(1):87–111, 2010.
10. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
11. A. Adriansyah, B. van Dongen, and W.M.P. van der Aalst. Conformance Checking using Cost-Based Fitness Analysis. In C.H. Chi and P. Johnson, editors, *IEEE International Enterprise Computing Conference (EDOC 2011)*, pages 55–64. IEEE Computer Society, 2011.
12. R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In *Sixth International Conference on Extending Database Technology*, volume 1377 of *Lecture Notes in Computer Science*, pages 469–483. Springer-Verlag, Berlin, 1998.
13. R. Agrawal and J.C. Shafer. Parallel Mining of Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962–969, 1996.
14. R.P. Jagadeesh Chandra Bose, W.M.P. van der Aalst, I. Zliobaite, and M. Pechenizkiy. Handling Concept Drift in Process Mining. In H. Mouratidis and C. Rolland, editors, *International Conference on Advanced Information Systems Engineering (Caise 2011)*, volume 6741 of *Lecture Notes in Computer Science*, pages 391–405. Springer-Verlag, Berlin, 2011.
15. C. Bratosin, N. Sidorova, and W.M.P. van der Aalst. Distributed Genetic Process Mining. In H. Ishibuchi, editor, *IEEE World Congress on Computational Intelligence (WCCI 2010)*, pages 1951–1958, Barcelona, Spain, July 2010. IEEE.

16. M. Cannataro, A. Congiusta, A. Pugliese, D. Talia, and P. Trunfio. Distributed Data Mining on Grids: Services, Tools, and Applications. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 34(6):2451–2465, 2004.
17. J. Carmona, J. Cortadella, and M. Kishinevsky. A Region-Based Algorithm for Discovering Petri Nets from Event Logs. In *Business Process Management (BPM2008)*, pages 358–373, 2008.
18. J.E. Cook and A.L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.
19. S. Goedertier, D. Martens, J. Vanthienen, and B. Baesens. Robust Process Discovery with Artificial Negative Events. *Journal of Machine Learning Research*, 10:1305–1340, 2009.
20. M. Hilbert and P. Lopez. The World's Technological Capacity to Store, Communicate, and Compute Information. *Science*, 332(60), 2011.
21. F.M. Maggi, A.J. Mooij, and W.M.P. van der Aalst. User-Guided Discovery of Declarative Process Models. In N. Chawla, I. King, and A. Sperduti, editors, *IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2011)*, pages 192–199, Paris, France, April 2011. IEEE.
22. J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. Byers. Big Data: The Next Frontier for Innovation, Competition, and Productivity. McKinsey Global Institute, 2011.
23. A.K. Alves de Medeiros, A.J.M.M. Weijters, and W.M.P. van der Aalst. Genetic Process Mining: An Experimental Evaluation. *Data Mining and Knowledge Discovery*, 14(2):245–304, 2007.
24. M. Montali, M. Pesic, W.M.P. van der Aalst, F. Chesani, P. Mello, and S. Storari. Declarative Specification and Verification of Service Choreographies. *ACM Transactions on the Web*, 4(1):1–62, 2010.
25. J. Munoz-Gama and J. Carmona. A Fresh Look at Precision in Process Conformance. In R. Hull, J. Mendling, and S. Tai, editors, *Business Process Management (BPM 2010)*, volume 6336 of *Lecture Notes in Computer Science*, pages 211–226. Springer-Verlag, Berlin, 2010.
26. J. Munoz-Gama and J. Carmona. Enhancing Precision in Process Conformance: Stability, Confidence and Severity. In N. Chawla, I. King, and A. Sperduti, editors, *IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2011)*, Paris, France, April 2011. IEEE.
27. A. Rozinat and W.M.P. van der Aalst. Conformance Checking of Processes Based on Monitoring Real Behavior. *Information Systems*, 33(1):64–95, 2008.
28. M. Sole and J. Carmona. Process Mining from a Basis of Regions. In J. Lilius and W. Penczek, editors, *Applications and Theory of Petri Nets 2010*, volume 6128 of *Lecture Notes in Computer Science*, pages 226–245. Springer-Verlag, Berlin, 2010.
29. A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.
30. J.M.E.M. van der Werf, B.F. van Dongen, C.A.J. Hurkens, and A. Serebrenik. Process Discovery using Integer Linear Programming. *Fundamenta Informaticae*, 94:387–412, 2010.
31. M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag, Berlin, 2007.