Discovering Petri Nets From Event Logs

W.M.P. van der Aalst and B.F. van Dongen

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, The Netherlands.
{W.M.P.v.d.Aalst,B.F.v.Dongen}@tue.nl

Abstract. As information systems are becoming more and more intertwined with the operational processes they support, multitudes of events are recorded by todays information systems. The goal of *process mining* is to use such event data to extract process related information, e.g., to automatically discover a process model by observing events recorded by some system or to check the conformance of a given model by comparing it with reality. In this article, we focus on *process discovery*, i.e., extracting a process model from an event log. We focus on Petri nets as a representation language, because of the concurrent and unstructured nature of real-life processes. The goal is to introduce several approaches to discover Petri nets from event data (notably the α-algorithm, state-based regions, and language-based regions). Moreover, important requirements for process discovery are discussed. For example, process mining is only meaningful if one can deal with *incompleteness* (only a fraction of all possible behavior is observed) and *noise* (one would like to abstract from infrequent random behavior). These requirements reveal significant challenges for future research in this domain.

Keywords: Process mining, Process discovery, Petri nets, Theory of regions

1 Introduction

Process mining provides a new means to improve processes in a variety of application domains [2, 41]. There are two main drivers for this new technology. On the one hand, more and more events are being recorded thus providing detailed information about the history of processes. Despite the omnipresence of event data, most organizations diagnose problems based on fiction rather than facts. On the other hand, vendors of Business Process Management (BPM) and Business Intelligence (BI) software have been promising miracles. Although BPM and BI technologies received lots of attention, they did not live up to the expectations raised by academics, consultants, and software vendors.

Process mining is an emerging discipline providing comprehensive sets of tools to provide fact-based insights and to support process improvements [2, 7]. This new discipline builds on process model-driven approaches and data mining. However, process mining is much more than an amalgamation of existing approaches. For example, existing data mining techniques are too data-centric to

provide a comprehensive understanding of the end-to-end processes in an organization. BI tools focus on simple dashboards and reporting rather than clear-cut business process insights. BPM suites heavily rely on experts modeling idealized to-be processes and do not help the stakeholders to understand the as-is processes.

Over the last decade *event data* has become readily available and process mining techniques have matured. Moreover, process mining algorithms have been implemented in various academic and commercial systems. Examples of commercial systems that support process mining are: *ARIS Process Performance Manager* by Software AG, *Disco* by Fluxicon, *Enterprise Visualization Suite* by Businesscape, *Interstage BPME* by Fujitsu, *Process Discovery Focus* by Iontas, *Reflect|one* by Pallas Athena, and *Reflect* by Futura Process Intelligence. Today, there is an active group of researchers working on process mining and it has become one of the "hot topics" in BPM research. Moreover, there is a huge interest from industry in process mining. This is illustrated by the recently released *Process Mining Manifesto* [41]. The manifesto is supported by 53 organizations and 77 process mining experts contributed to it. The manifesto has been translated into a dozen languages (http://www.win.tue.nl/ieeetfpm/). The active contributions from end-users, tool vendors, consultants, analysts, and researchers illustrate the growing relevance of process mining as a bridge between data mining and business process modeling. Moreover, more and more software vendors started adding process mining functionality to their tools. The authors have been involved in the development of the open-source process mining tool *ProM* right from the start [11, 56, 57]. ProM is widely used all over the globe and provides an easy starting point for practitioners, students, and academics.

Whereas it is easy to discover sequential processes, it is very challenging to discover concurrent processes, especially in the context of noisy and incomplete event logs. Given the concurrent nature of most real-life processes, Petri nets are an obvious candidate to represent discovered processes. Moreover, most real-life processes are not nicely block-structured, therefore, the graph based nature of Petri nets is more suitable than notations that enforce more structure.

The article is based on a lecture given at the *Advanced Course on Petri nets* in Rostock, Germany (September 2010). The practical relevance of process discovery and the suitability of Petri net as a basic representation for concurrent processes motivated us to write this tutorial.

Figure 1 illustrates the concept of process discovery using a small example. The figure shows an abstraction of an event log. There are 1391 cases, i.e., process instances. Each case is described as a sequence of activities, i.e., a trace. In this particular log there are 21 different traces. For example, trace $\langle a, c, d, e, h\rangle$ occurs 455 times, i.e., there are 455 cases for which this sequence of activities was executed. The challenge is to discover a Petri net given such an event log. A discovery algorithm such as the α-algorithm [9] is able to discover the Petri net shown in Figure 1.

Process discovery is a challenging problem because one cannot assume that all possible sequences are indeed present. Consider for example the event log

#	trace
455	acdeh
191	abdeg
177	adceh
144	abdeh
111	acdeg
82	adceg
56	adbeh
47	acdefdbeh
38	adbeg
33	acdefbdeh
14	acdefbdeg

#	trace
11	acdefdbeg
9	adcefcdeh
8	adcefdbeh
5	adcefbdeg
3	acdefbdefdbeg
2	adcefdbeg
2	adcefbdefbdeg
1	adcefdbefbdeh
1	adbefbdefdbeg
1	adcefdbefcdefdbeg

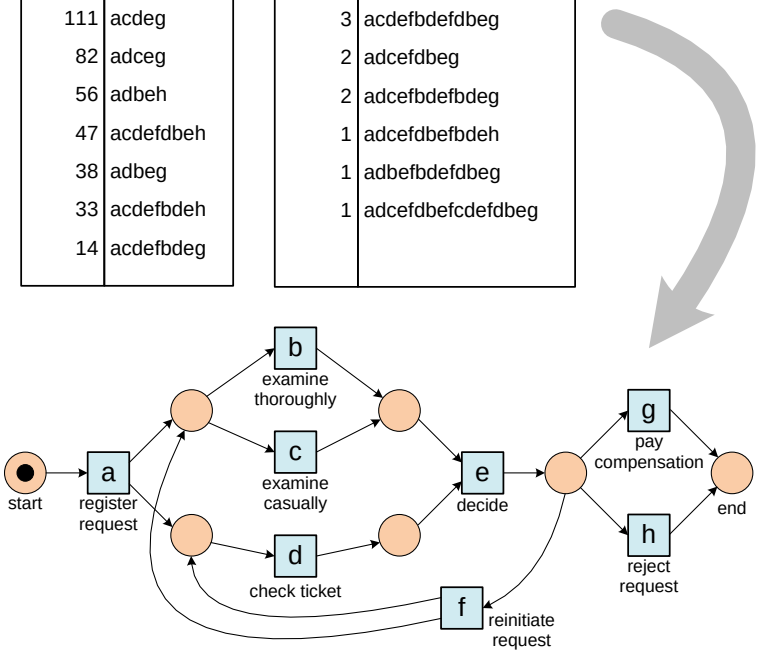


Fig. 1. A Petri net discovered from an event log containing 1391 cases.

shown in Figure 1. If we randomly take 500 cases from the set of 1391 cases, we would like to discover "more or less" the same model. Note that there are several traces that appear only once in the log. Many of these will disappear when considering a log with only 500 cases. Also note that the process model discovered by the α-algorithm allows for more traces than the ones depicted in Figure 1, e.g., $\langle a, d, c, e, f, d, b, e, f, c, d, e, h \rangle$ is possible according to the process model but does not occur in the event log. This illustrates that event logs tend to be *far from complete*, i.e., only a small subset of all possible behavior can be observed because the number of variations is larger than the number of instances observed.

The process model in Figure 1 is rather simple. Real-life processes will consist of dozens or even hundreds of different activities. Moreover, some behaviors will be very infrequent compared to others. Such rare behaviors can be seen as

noise (e.g., exceptions). Typically, it is undesirable and also unfeasible to capture frequent and infrequent behavior in a single diagram.

Process discovery techniques need to be able to deal with noise and incompleteness. This makes process mining *very different from synthesis*. Classical synthesis techniques aim at creating a model that captures the given behavior *precisely*. For example, classical language-based region techniques [14, 17, 19, 28, 42, 43, 45] distill a Petri net from a (possibly infinite) language, such that the behavior of the Petri net is only minimally more than the given language. In classical state-based region theory [13, 15, 23, 24, 26, 27, 35] on the other hand, a transition system is used to synthesize a Petri net of which the behavior is *bisimilar* with the given transition system. Intuitively two models are bisimilar if they can match each other's moves, i.e., they cannot be distinguished from one another by an observer [36]. In terms of mining this implies that the naïvely synthesized Petri net cannot generalize beyond the example traces seen.

Process discovery techniques need to balance four criteria: *fitness* (the discovered model should allow for the behavior seen in the event log), *precision* (the discovered model should not allow for behavior completely unrelated to what was seen in the event log), *generalization* (the discovered model should generalize the example behavior seen in the event log), and *simplicity* (the discovered model should be as simple as possible). This makes process discovery a challenging and highly relevant topic.

The remainder of this article is organized as follows. Section 2 introduces the process mining spectrum showing that process discovery is an essential ingredient for process analysis based on facts rather than fiction. Section 3 presents preliminaries and formalizes the process discovery task. The α-algorithm is presented in Section 4. Section 5 discusses the main challenges related to process mining. In Section 6, we compare process discovery with region theory in more detail. This section shows that classical approaches cannot deal with particular requirements essential for process mining. Then, in sections 7 and 8, we show how region theory can be adapted to deal with these requirements. Both state-based regions and language-based regions are considered. All approaches described in this article are supported by *ProM*, the leading open-source process mining framework. ProM is described in Section 9. Section 10 ends this article with some conclusions and challenges that remain.

2 Process Mining

Process mining is an important tool for modern organizations that need to manage non-trivial operational processes. On the one hand, there is an incredible growth of event data [44]. On the other hand, processes and information need to be aligned perfectly in order to meet requirements related to compliance, efficiency, and customer service. *Process mining is much broader than just control-flow discovery*, i.e., discovering a Petri net from a multi-set of traces. Therefore, we start by providing an overview of the process mining spectrum.

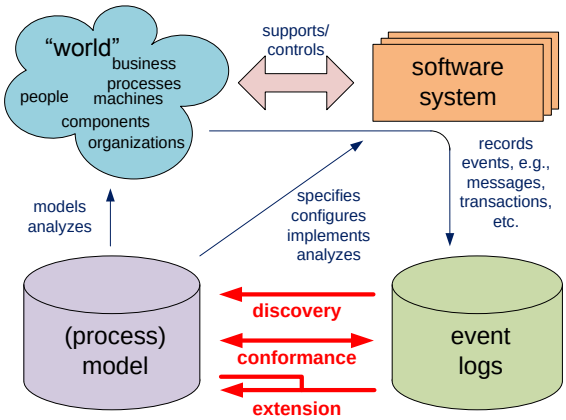


Fig. 2. Positioning of the three main types of process mining: *discovery*, *conformance*, and *enhancement*.

Event logs can be used to conduct three types of process mining as shown in Figure 2 [2, 7].

The first type of process mining is *discovery*. A discovery technique takes an event log and produces a model without using any a-priori information. An example is the α-algorithm [9] that will be described in Section 4. This algorithm takes an event log and produces a Petri net explaining the behavior recorded in the log. For example, given sufficient example executions of the process shown in Figure 1, the α-algorithm is able to automatically construct the Petri net without using any additional knowledge. If the event log contains information about resources, one can also discover resource-related models, e.g., a social network showing how people work together in an organization.

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. For instance, there may be a process model indicating that purchase orders of more than one million Euro require two checks. Analysis of the event log will show whether this rule is followed or not. Another example is the checking of the so-called "four-eyes" principle stating that particular activities should not be executed by one and the same person. By scanning the event log using a model specifying these requirements, one can discover potential cases of fraud. Hence, conformance checking may be used to detect, locate and explain deviations, and to measure the severity of these deviations. An example is the conformance checking algorithm described in [51]. Given the model shown in Figure 1 and a corresponding event log, this algorithm can quantify and diagnose deviations. In [4] another approach based on creating *alignments* is presented. An alignment is *optimal* if it relates the trace in the log to a most similar path in the model.

After creating optimal alignments, all behavior in the log can be related to the model.

The third type of process mining is *enhancement*. Here, the idea is to extend or improve an existing process model using information about the actual process recorded in some event log. Whereas conformance checking measures the alignment between model and reality, this third type of process mining aims at changing or extending the a-priori model. One type of enhancement is *repair*, i.e., modifying the model to better reflect reality. For example, if two activities are modeled sequentially but in reality can happen in any order, then the model may be corrected to reflect this. Another type of enhancement is *extension*, i.e., adding a new perspective to the process model by cross-correlating it with the log. An example is the extension of a process model with performance data. For instance, Figure 1 can be extended with information about resources, decision rules, quality metrics, etc.

The Petri net in Figure 1 only shows the control-flow. However, when extending process models, additional perspectives need to be added. Moreover, discovery and conformance techniques are not limited to control-flow. For example, one can discover a social network and check the validity of some organizational model using an event log. Hence, orthogonal to the three types of mining (discovery, conformance, and enhancement), different perspectives can be identified. The *organizational perspective* focuses on information about resources hidden in the log, i.e., which actors (e.g., people, systems, roles, and departments) are involved and how are they related. The goal is to either structure the organization by classifying people in terms of roles and organizational units or to show the social network. The *time perspective* is concerned with the timing and frequency of events. When events bear timestamps it is possible to discover bottlenecks, measure service levels, monitor the utilization of resources, and predict the remaining processing time of running cases.

3 Process Discovery: Preliminaries and Purpose

In this section, we describe the goal of process discovery. In order to do this, we present a particular format for logging events and a particular process modeling language (i.e., Petri nets). Based on this we sketch various process discovery approaches.

3.1 Event Logs

The goal of process mining is to extract knowledge about a particular (operational) process from event logs, i.e., process mining describes a family of *a-posteriori* analysis techniques exploiting the information recorded in audit trails, transaction logs, databases, etc. Typically, these approaches assume that it is possible to *sequentially record events* such that each event refers to an *activity* (i.e., a well-defined step in the process) and is related to a particular *case* (i.e., a process instance). Furthermore, some mining techniques use additional

information such as the performer or *originator* of the event (i.e., the person / resource executing or initiating the activity), the *timestamp* of the event, or *data elements* recorded with the event (e.g., the size of an order).

To clarify the notion of an event log consider Table 1 which shows a fragment of some event log. Only two traces are shown, both containing four events. Each event has a unique id and several properties. For example event 35654423 belongs to case $x123$ and is an instance of activity a that occurred on December 30th at 11.02, was executed by John, and cost 300 euros. The second trace (case $x128$) starts with event 35655526 and also refers to an instance of activity a. The

Table 1. A fragment of some event log.

case id	event id	properties				
		timestamp	activity	resource	cost	...
x123	35654423	30-12-2011:11.02	a	John	300	...
x123	35654424	30-12-2011:11.06	b	John	400	...
x123	35654425	30-12-2011:11.12	c	John	100	...
x123	35654426	30-12-2011:11.18	d	John	400	...
x128	35655526	30-12-2011:16.10	a	Ann	300	...
x128	35655527	30-12-2011:16.14	c	John	450	...
x128	35655528	30-12-2011:16.26	b	Pete	350	...
x128	35655529	30-12-2011:16.36	d	Ann	300	...
...	...	...	...	...	...	...

information depicted in Table 1 is the typical event data that can be extracted from today's systems.

Systems store events in very different ways. Process-aware information systems (e.g., workflow management systems) provide dedicated audit trails. In other systems, this information is typically scattered over several tables. For example, in a hospital events related to a particular patient may be stored in different tables and even different systems. For many applications of process mining, one needs to extract event data from different sources, merge these data, and convert the result into a suitable format. We advocate the use of the so-called *XES (eXtensible Event Stream)* format that can be read directly by ProM ([5, 57]). XES is the successor of MXML. Based on many practical experiences with MXML, the XES format has been made less restrictive and truly extendible. In September 2010, the format was adopted by the IEEE Task Force on Process Mining. The format is supported by tools such as ProM (as of version 6), Nitro, XESame, and OpenXES. See www.xes-standard.org for detailed information about the standard. XES is able to store the information shown in Table 1. Most of this information is optional, i.e., if it is there, it can be used for process mining, but it is not necessary for control-flow discovery.

In this article, we focus on control-flow discovery. Therefore, we only consider the activity column in Table 1. This means that an event is linked to a case (process instance) and an activity, and no further attributes are needed. Events are ordered (per case), but do not need to have explicit timestamps. This allows us to use the following simplified definition of an event log.

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

The first four events in Table 1 form a trace $\langle a, b, c, d \rangle$. This trace represents the path followed by case $x123$. The second case ($x128$) can be represented by the trace $\langle a, c, b, d \rangle$. Note that there may be multiple cases that have the same trace. Therefore, an event log is defined as a *multi-set* of traces.

A *multi-set* (also referred to as *bag*) is like a set where each element may occur multiple times. For example, $[horse, cow^5, duck^2]$ is the multi-set with eight elements: one horse, five cows and two ducks. $\mathbb{B}(X)$ is the set of multi-sets (bags) over X. We assume the usual operators on multi-sets, e.g., $X \cup Y$ is the union of X and Y, $X \setminus Y$ is the difference between X and Y, $x \in X$ tests if x appears in X, and $X \leq Y$ evaluates to true if X is contained in Y. For example, $[horse, cow^2] \cup [horse^2, duck^2] = [horse^3, cow^2, duck^2]$, $[horse^3, cow^4] \setminus [cow^2] = [horse^3, cow^2]$, $[horse, cow^2] \leq [horse^2, cow^3]$, and $[horse^3, cow^1] \not\leq [horse^2, cow^2]$. Note that sets can be considered as bags having only one instance of every element. Hence, we can mix sets and bags, e.g., $\{horse, cow\} \cup [horse^2, cow^3] = [horse^3, cow^4]$.

For practical applications of process mining it is essential to differentiate between traces that are infrequent or even unique (multiplicity of 1) and traces that are frequent. Therefore, an event log is a *multi-set of traces* rather than an ordinary set. However, in this article we focus on the foundations of process discovery thereby often abstracting from noise and frequencies. See [2] for techniques that take frequencies into account. This book also describes various case studies showing the importance of multiplicities.

In the remainder, we will use the following example log: $L_1 = [\langle a, b, c, d \rangle^5, \langle a, c, b, d \rangle^8, \langle a, e, d \rangle^9]$. L_1 contains information about 22 cases; five cases following trace $\langle a, b, c, d \rangle$, eight cases following trace $\langle a, c, b, d \rangle$, and nine cases following trace $\langle a, e, d \rangle$. Note that such a simple representation can be extracted from sources such as Table 1, MXML, XES, or any other format that links events to cases and activities.

3.2 Petri Nets

The goal of process discovery is to distil a process model from some event log. Here we use *Petri nets* [50] to represent such models. In fact, we extract a subclass of Petri nets known as *workflow nets* (WF-nets) [1].

Definition 2. *An* Petri net *is a tuple (P, T, F) where:*

1. *P is a finite set of* places*,*

2. T *is a finite set of* transitions *such that* $P \cap T = \emptyset$*, and*
3. $F \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs, called the* flow relation*.*

An example Petri net is shown in Figure 3. This Petri net has six places represented by circles and four transitions represented by squares. Places may contain tokens. For example, in Figure 3 both $p1$ and $p6$ contain one token, $p3$ contains two tokens, and the other places are empty. The state, also called *marking*, is the distribution of tokens over places. A *marked* Petri net is a pair (N, M), where $N = (P, T, F)$ is a Petri net and where $M \in \mathbb{B}(P)$ is a bag over P denoting the *marking* of the net. The initial marking of the Petri net shown in Figure 3 is $[p1, p3^2, p6]$. The set of all marked Petri nets is denoted $\mathcal{N}$.

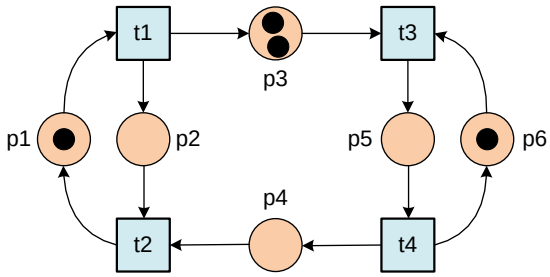


Fig. 3. A Petri net with six places ($p1$, $p2$, $p3$, $p4$, $p5$, and $p6$) and four transitions ($t1$, $t2$, $t3$, and $t4$).

Let $N = (P, T, F)$ be a Petri net. Elements of $P \cup T$ are called *nodes*. A node x is an *input node* of another node y iff there is a directed arc from x to y (i.e., $(x, y) \in F$). Node x is an *output node* of y iff $(y, x) \in F$. For any $x \in P \cup T$, ${}^\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$. In Figure 3, ${}^\bullet t3 = \{p3, p6\}$ and $t3^\bullet = \{p5\}$.

The dynamic behavior of such a marked Petri net is defined by the so-called *firing rule*. A transition is *enabled* if each of its input places contains a token. An enabled transition can *fire* thereby consuming one token from each input place and producing one token for each output place.

Definition 3 (Firing rule). *Let* (N, M) *be a marked Petri net with* $N = (P, T, F)$*. Transition* $t \in T$ *is* enabled*, denoted* $(N, M)[t\rangle$*, iff* ${}^\bullet t \leq M$*. The* firing rule $\_\,[\_\rangle\,\_ \subseteq \mathcal{N} \times T \times \mathcal{N}$ *is the smallest relation satisfying for any* $(N, M) \in \mathcal{N}$ *and any* $t \in T$*,* $(N, M)[t\rangle \Rightarrow (N, M)\,[t\rangle\,(N, (M \setminus {}^\bullet t) \cup t^\bullet)$*.*

In the marking shown in Figure 3, both $t1$ and $t3$ are enabled. The other two transitions are not enabled because at least one of the input places is empty. If $t1$ fires, one token is consumed (from $p1$) and two tokens are produced (one for $p2$ and one for $p3$). Formally, $(N, [p1, p3^2, p6])\,[t1\rangle\,(N, [p2, p3^3, p6])$. So the resulting marking is $[p2, p3^3, p6]$. If $t3$ fires in the initial state, two tokens are

consumed (one from $p3$ and one from $p6$) and one token is produced (for $p5$). Formally, $(N, [p1, p3^2, p6])\ [t3\rangle\ (N, [p1, p3, p5])$.

Let (N, M_0) with $N = (P, T, F)$ be a marked P/T net. A sequence $\sigma \in T^*$ is called a *firing sequence* of (N, M_0) iff, for some natural number $n \in \mathbb{N}$, there exist markings $M_1, \ldots, M_n$ and transitions $t_1, \ldots, t_n \in T$ such that $\sigma = \langle t_1 \ldots t_n \rangle$ and, for all i with $0 \leq i < n$, $(N, M_i)[t_{i+1}\rangle$ and $(N, M_i)\ [t_{i+1}\rangle\ (N, M_{i+1})$.

Let (N, M_0) be the marked Petri net shown in Figure 3, i.e., $M_0 = [p1, p3^2, p6]$. The empty sequence $\sigma = \langle\ \rangle$ is enabled in (N, M_0). The sequence $\sigma = \langle t1, t3 \rangle$ is also enabled and results in marking $[p2, p3^2, p5]$. Another possible firing sequence is $\sigma = \langle t3, t4, t3, t1, t4, t3, t2, t1 \rangle$. A marking M is *reachable* from the initial marking M_0 iff there exists a sequence of enabled transitions whose firing leads from M_0 to M. The set of reachable markings of (N, M_0) is denoted $[N, M_0\rangle$.

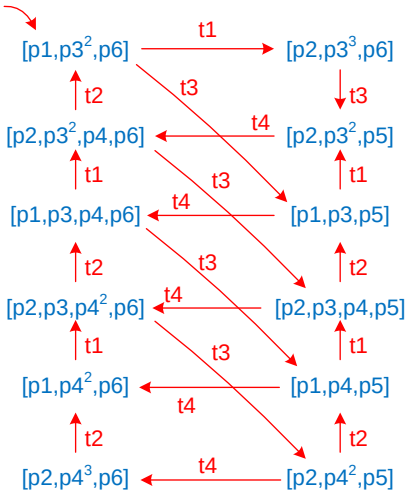


Fig. 4. The reachability graph of the marked Petri net shown in Figure 3.

For the marked Petri net shown in Figure 3 there are 12 reachable states. These states can be computed using the so-called *reachability graph* shown in Figure 4. All nodes correspond to reachable markings and each arc corresponds to the firing of a particular transition. Any path in the reachability graph corresponds to a possible firing sequence. For example, using Figure 4 is is easy to see that $\langle t3, t4, t3, t1, t4, t3, t2, t1 \rangle$ is indeed possible and results in $[p2, p3, p4, p5]$. A marked net may be unbounded, i.e., have an infinite number or reachable states. In this case, the reachability graph is infinitely large, but one can still construct the so-called coverability graph [50].

3.3 Workflow Nets

For process discovery, we look at processes that are instantiated multiple times, i.e., the same process is executed for multiple cases. For example, the process of handling insurance claims may be executed for thousands or even millions of claims. Such processes have a clear starting point and a clear ending point. Therefore, the following subclass of Petri nets (WF-nets) is most relevant for process discovery.

Definition 4 (Workflow nets). *Let $N = (P, T, F)$ be a Petri net and $\bar{t}$ a fresh identifier not in $P \cup T$. N is a* workflow net *(WF-net) iff:*

1. *object creation: P contains an input place i (also called source place) such that ${}^\bullet i = \emptyset$,*
2. *object completion: P contains an output place o (also called sink place) such that $o^\bullet = \emptyset$,*
3. *connectedness: $\bar{N} = (P, T \cup \{\bar{t}\}, F \cup \{(o, \bar{t}), (\bar{t}, i)\})$ is strongly connected, i.e., there is a directed path between any pair of nodes in $\bar{N}$.*

Clearly, Figure 3 is not a WF-net because a source and sink place are missing. Figure 5 shows an example of a WF-net: ${}^\bullet start = \emptyset$, $end^\bullet = \emptyset$, and every node is on a path from *start* to *end*.

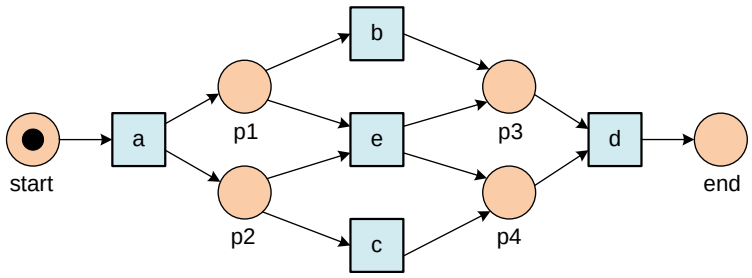


Fig. 5. A workflow net with source place $i = start$ and sink place $o = end$.

The Petri net depicted in Figure 1 is another example of a WF-net. Not every WF-net represents a correct process. For example, a process represented by a WF-net may exhibit errors such as deadlocks, tasks which can never become active, livelocks, garbage being left in the process after termination, etc. Therefore, we define the following correctness criterion.

Definition 5 (Soundness). *Let $N = (P, T, F)$ be a WF-net with input place i and output place o. N is* sound *iff:*

1. *safeness: $(N, [i])$ is safe, i.e., places cannot hold multiple tokens at the same time,*

2. *proper completion: for any marking* $M \in [N, [i]\rangle$, $o \in M$ *implies* $M = [o]$,
3. *option to complete: for any marking* $M \in [N, [i]\rangle$, $[o] \in [N, M\rangle$, *and*
4. *absence of dead tasks:* $(N, [i])$ *contains no dead transitions (i.e., for any* $t \in T$, *there is a firing sequence enabling* t*).*

The WF-nets shown in figures 5 and 1 are sound. Soundness can be verified using standard Petri-net-based analysis techniques. In fact soundness corresponds to liveness and safeness of the corresponding short-circuited net [1]. This way efficient algorithms and tools can be applied. An example of a tool tailored towards the analysis of WF-nets is Woflan [55]. This functionality is also embedded in our process mining tool ProM [5].

3.4 Problem Definition and Approaches

After introducing events logs and WF-nets, we can define the main goal of process discovery.

Definition 6 (Process discovery). *Let* L *be an event log over* A, *i.e.,* $L \in \mathbb{B}(A^*)$. *A* process discovery algorithm *is a function* γ *that maps any log* L *onto a Petri net* $\gamma(L) = (N, M)$. *Ideally,* N *is a sound WF-net and all traces in* L *correspond to possible firing sequences of* (N, M).

The goal is to find a process model that can "replay" all cases recorded in the log, i.e., all traces in the log are possible firing sequences of the discovered WF-net. Assume that $L_1 = [\langle a, b, c, d\rangle^5, \langle a, c, b, d\rangle^8, \langle a, e, d\rangle^9]$. In this case the WF-net shown in Figure 5 is a good solution. All traces in L_1 correspond to firing sequences of the WF-net and vice versa. Throughout this article, we use L_1 as an example log. Note that it may be possible that some of the firing sequences of the discovered WF-net do not appear in the log. This is acceptable as one cannot assume that all possible sequences have been observed. For example, if there is a loop, the number of possible firing sequences is infinite. Even if the model is acyclic, the number of possible sequences may be enormous due to choices and parallelism. Later in this article, we will discuss the quality of discovered models in more detail.

Since the mid-nineties several groups have been working on techniques for process mining [7, 9, 10, 25, 29, 32, 33, 58], i.e., discovering process models based on observed events. In [6] an overview is given of the early work in this domain. The idea to apply process mining in the context of workflow management systems was introduced in [10]. In parallel, Datta [29] looked at the discovery of business process models. Cook et al. investigated similar issues in the context of software engineering processes [25]. Herbst [40] was one of the first to tackle more complicated processes, e.g., processes containing duplicate tasks.

Most of the classical approaches have problems dealing with concurrency. The α-algorithm [9] is an example of a simple technique that takes concurrency as a starting point. However, this simple algorithm has problems dealing with complicated routing constructs and noise (like most of the other approaches

described in literature). In [32, 33] a more robust but less precise approach is presented.

Recently, people started using the "theory of regions" to process discovery. There are two approaches: state-based regions and language-based regions. State-based regions can be used to convert a transition system into a Petri net [13, 15, 23, 24, 26, 27, 35]. Language-based regions add places as long as it is still possible to replay the log [14, 17, 19, 28, 42, 43].

More from a theoretical point of view, the process discovery problem is related to the work discussed in [12, 37, 38, 49]. In these papers the limits of inductive inference are explored. For example, in [38] it is shown that the computational problem of finding a minimum finite-state acceptor compatible with given data is NP-hard. Several of the more generic concepts discussed in these papers can be translated to the domain of process mining. It is possible to interpret the problem described in this article as an inductive inference problem specified in terms of rules, a hypothesis space, examples, and criteria for successful inference. The comparison with literature in this domain raises interesting questions for process mining, e.g., how to deal with negative examples (i.e., suppose that besides log L there is a log L' of traces that are not possible, e.g., added by a domain expert). However, despite the relations with the work described in [12, 37, 38, 49] there are also many differences, e.g., we are mining at the net level rather than sequential or lower level representations (e.g., Markov chains, finite state machines, or regular expressions), tackle concurrency, and do not assume negative examples or complete logs.

The above approaches assume that there is no noise or infrequent behavior. For approaches dealing with these problems we refer to the work done by Christian Günther [39], Ton Weijters [58], and Ana Karla Alves de Medeiros [47].

4 α-Algorithm

After introducing the process discovery problem and providing an overview of approaches described in literature, we focus on the α-algorithm [9]. The α-algorithm is not intended as a practical mining technique as it has problems with noise, infrequent/incomplete behavior, and complex routing constructs. Nevertheless, it provides a good introduction into the topic. The α-algorithm is very simple and many of its ideas have been embedded in more complex and robust techniques. Moreover, it was the first algorithm to really address the discovery of concurrency.

4.1 Basic Idea

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

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

Consider for example $L_1 = [\langle a, b, c, d \rangle^5, \langle a, c, b, d \rangle^8, \langle a, e, d \rangle^9]$. $c >_{L_1} d$ because d directly follows c in trace $\langle a, b, c, d \rangle$. However, $d \not>_{L_1} c$ because c never directly follows d in any trace in the log.

$>_{L_1} = \{(a, b), (a, c), (a, e), (b, c), (c, b), (b, d), (c, d), (e, d)\}$ contains all pairs of activities in a "directly follows" relation. $c \rightarrow_{L_1} d$ because sometimes d directly follows c and never the other way around ($c >_{L_1} d$ and $d \not>_{L_1} c$). $\rightarrow_{L_1} = \{(a, b), (a, c), (a, e), (b, d), (c, d), (e, d)\}$ contains all pairs of activities in a "causality" relation. $b \|_{L_1} c$ because $b >_{L_1} c$ and $c >_{L_1} b$, i.e, sometimes c follows b and sometimes the other way around. $\|_{L_1} = \{(b, c), (c, b)\}$. $b \#_{L_1} e$ because $b \not>_{L_1} e$ and $e \not>_{L_1} b$. $\#_{L_1} = \{(a, a), (a, d), (b, b), (b, e), (c, c), (c, e), (d, a), (d, d), (e, b), (e, c), (e, e)\}$. Note that for any log L over A and $x, y \in A$: $x \rightarrow_L y$, $y \rightarrow_L x$, $x \#_L y$, or $x \|_L y$.

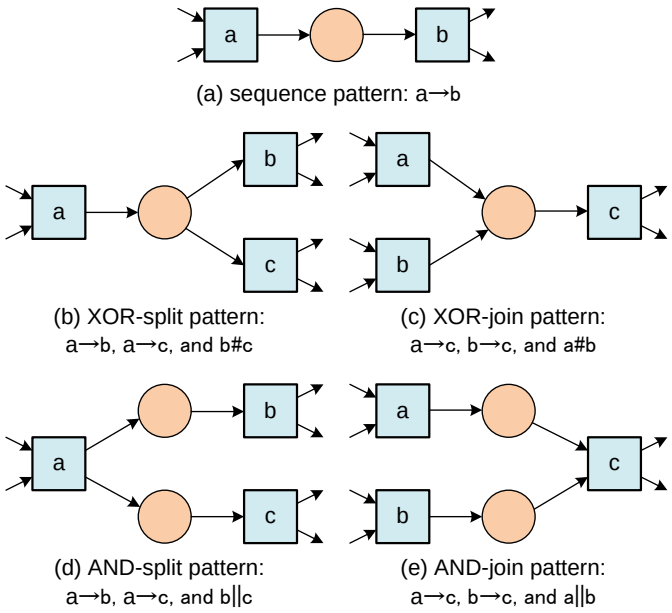


Fig. 6. Typical process patterns and the footprints they leave in the event log.

The log-based ordering relations can be used to discover patterns in the corresponding process model as is illustrated in Figure 6. If a and b are in sequence, the log will show $a >_L b$. 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 Figure 6(b-c). If $a \rightarrow_L c$, $b \rightarrow_L c$, and $a \#_L b$, then this suggests that after the occurrence of either a or b, c should happen. Figure 6(d-e) shows the so-called AND-split and AND-join patterns. If $a \rightarrow_L b$, $a \rightarrow_L c$, and $b \|_L c$, then it appears that after a both b and c can be executed in parallel (AND-split pattern). If $a \rightarrow_L c$, $b \rightarrow_L c$, and $a \|_L b$, then it appears that c needs to synchronize a and b (AND-join pattern).

Figure 6 only shows simple patterns and does not present the additional conditions needed to extract the patterns. However, the figure nicely illustrates the basic idea.

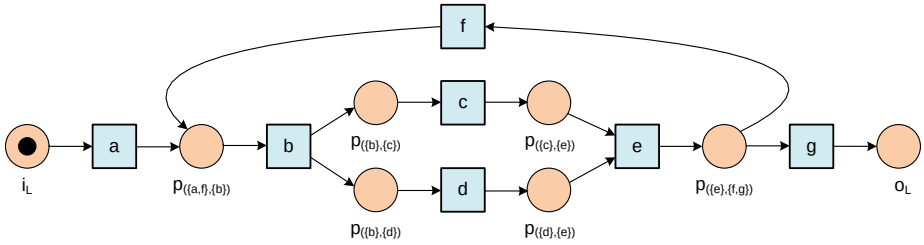


Fig. 7. WF-net N_2 derived from $L_2 = [\langle a,b,c,d,e,f,b,d,c,e,g\rangle, \langle a,b,d,c,e,g\rangle, \langle a,b,c,d,e,f,b,c,d,e,f,b,d,c,e,g\rangle]$.

Consider for example WF-net N_2 depicted in Figure 7 and the log event log $L_2 = [\langle a,b,c,d,e,f,b,d,c,e,g\rangle, \langle a,b,d,c,e,g\rangle,$
$\langle a,b,c,d,e,f,b,c,d,e,f,b,d,c,e,g\rangle]$. The α-algorithm constructs WF-net N_2 based on L_2. Note that the patterns in the model indeed match the log-based ordering relations extracted from the event log. Consider for example the process fragment involving b, c, d, and e. Obviously, this fragment can be constructed based on $b \rightarrow_{L_2} c$, $b \rightarrow_{L_2} d$, $c \|_{L_2} d$, $c \rightarrow_{L_2} e$, and $d \rightarrow_{L_2} e$. The choice following e is revealed by $e \rightarrow_{L_2} f$, $e \rightarrow_{L_2} g$, and $f \#_{L_2} g$. Etc.

Another example is shown in Figure 8. WF-net N_3 can be derived from $L_3 = [\langle a,c,d\rangle^{45}, \langle b,c,d\rangle^{42}, \langle a,c,e\rangle^{38}, \langle b,c,e\rangle^{22}]$. Note that here there are two start and two end activities. These can be found easily by looking for the first and last activities in traces.

4.2 Algorithm

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

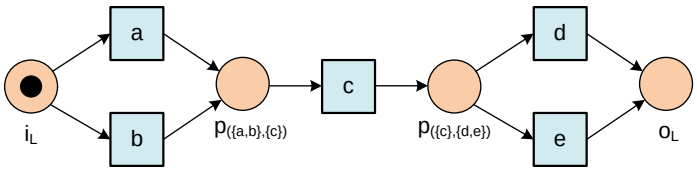


Fig. 8. WF-net N_3 derived from $L_3 = [\langle a, c, d\rangle^{45}, \langle b, c, d\rangle^{42}, \langle a, c, e\rangle^{38}, \langle b, c, e\rangle^{22}]$.

Definition 8 (α-algorithm). *Let L be an event log over T. $\alpha(L)$ is defined as follows.*

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

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

The basic idea for finding $p_{(A,B)}$ is shown in Figure 9. All elements of A should have causal dependencies with all elements of B, i.e., for any $(a, b) \in A \times B$: $a \rightarrow_L b$. Moreover, the elements of A should never follow any of the other elements, i.e., for any $a_1, a_2 \in A$: $a_1 \#_L a_2$. A similar requirement holds for B.

Let us consider $L_1 = [\langle a, b, c, d\rangle^5, \langle a, c, b, d\rangle^8, \langle a, e, d\rangle^9]$. Clearly $A = \{a\}$ and $B = \{b, e\}$ meet the requirements stated in Step 4. Also note that $A' = \{a\}$ and $B' = \{b\}$ meet the same requirements. X_L is the set of all such pairs that meet the requirements just mentioned. In this case, $X_{L_1} = \{(\{a\}, \{b\}), (\{a\}, \{c\}), (\{a\}, \{e\}), (\{a\}, \{b, e\}), (\{a\}, \{c, e\}), (\{b\}, \{d\}), (\{c\}, \{d\}), (\{e\}, \{d\}), (\{b, e\}, \{d\}), (\{c, e\}, \{d\})\}$. If one would insert a place for any element in X_{L_1} there would be too many places. Therefore, only the "maximal pairs" (A, B) should be included. Note that for any pair $(A, B) \in X_L$, non-empty set $A' \subseteq A$, and non-empty set $B' \subseteq B$, it is implied that $(A', B') \in X_L$. In Step 5 all non-maximal pairs are removed. So $Y_{L_1} = \{(\{a\}, \{b, e\}), (\{a\}, \{c, e\}), (\{b, e\}, \{d\}), (\{c, e\}, \{d\})\}$.

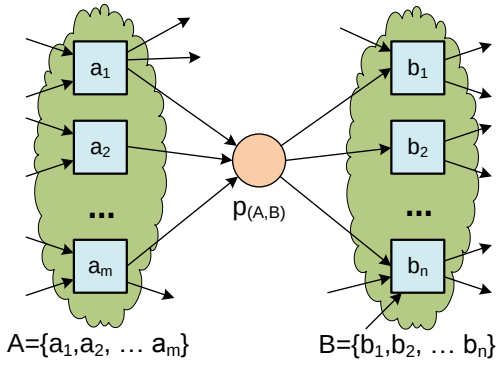


Fig. 9. Place $p_{(A,B)}$ connects the transitions in set A to the transitions in set B.

Every element of $(A, B) \in Y_L$ corresponds to a place $p_{(A,B)}$ connecting transitions A to transitions B. In addition P_L also contains a unique source place i_L and a unique sink place o_L (cf. Step 6).

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

Thus far we presented three logs and three WF-nets. Clearly $\alpha(L_2) = N_2$, and $\alpha(L_3) = N_3$. In figures 7 and 8 the places are named based on the sets Y_{L_2} and Y_{L_3}. Moreover, $\alpha(L_1) = N_1$ modulo renaming of places (because different place names are used in Figure 5). These examples show that the α-algorithm is indeed able to discover WF-nets based event logs.

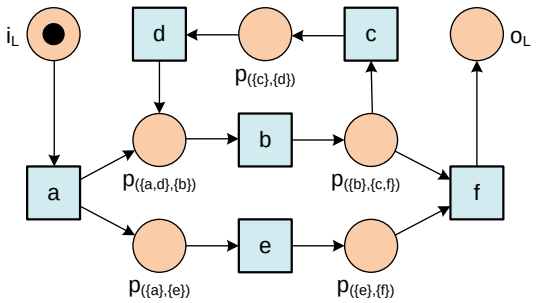


Fig. 10. WF-net N_4 derived from $L_4 = [\langle a, b, e, f\rangle^2, \langle a, b, e, c, d, b, f\rangle^3, \langle a, b, c, e, d, b, f\rangle^2, \langle a, b, c, d, e, b, f\rangle^4, \langle a, e, b, c, d, b, f\rangle^3]$.

Figure 10 shows another example. WF-net N_4 can be derived from $L_4 = [\langle a, b, e, f\rangle^2, \langle a, b, e, c, d, b, f\rangle^3, \langle a, b, c, e, d, b, f\rangle^2, \langle a, b, c, d, e, b, f\rangle^4, \langle a, e, b, c, d, b, f\rangle^3]$, i.e., $\alpha(L_4) = N_4$.

The WF-net in Figure 1 is discovered when applying the α-algorithm to the event log in the same figure.

4.3 Limitations

In [9] it was shown that the α-algorithm is able to discover a large class of WF-nets if one assumes that the log is *complete* with respect to the log-based ordering relation $>_L$. This assumption implies that, for any event log L, $a >_L b$ if a can be directly followed by b. We revisit the notion of completeness later in this article.

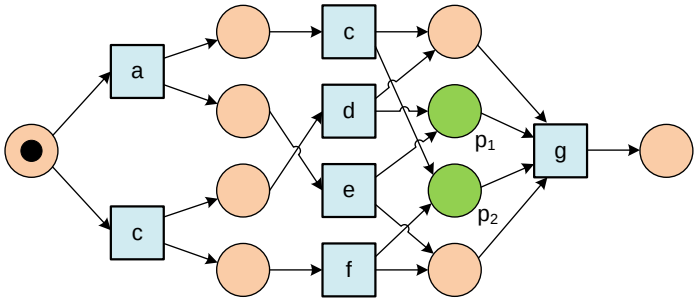


Fig. 11. WF-net N_5 derived from $L_5 = [\langle a, c, e, g\rangle^2, \langle a, e, c, g\rangle^3, \langle b, d, f, g\rangle^2, \langle b, f, d, g\rangle^4]$.

Even if we assume that the log is complete, the α-algorithm has some problems. There are many different WF-nets that have the same possible behavior, i.e., two models can be structurally different but trace equivalent. Consider for example $L_5 = [\langle a, c, e, g\rangle^2, \langle a, e, c, g\rangle^3, \langle b, d, f, g\rangle^2, \langle b, f, d, g\rangle^4]$. $\alpha(L_5)$ is shown in Figure 11. Although the model is able to generate the observed behavior, the resulting WF-net is needlessly complex. Two of the input places of g are redundant, i.e., they can be removed without changing the behavior. The places denoted as p_1 and p_2 are so-called implicit places and can be removed without allowing for more traces. In fact, Figure 11 shows only one of many possible trace equivalent WF-nets.

The original α-algorithm has problems dealing with short loops, i.e., loops of length 1 or 2. This is illustrated by WF-net N_6 in Figure 12 which shows the result of applying the basic algorithm to $L_6 = [\langle a, c\rangle^2, \langle a, b, c\rangle^3, \langle a, b, b, c\rangle^2]$. It is easy to see that the model does not allow for $\langle a, c\rangle$ and $\langle a, b, b, c\rangle$. In fact, in N_6, transition b needs to be executed precisely once and there is an implicit place connecting a and c. This problem can be addressed easily as shown in [46]. Using an improved version of the α-algorithm one can discover the WF-net shown in Figure 13.

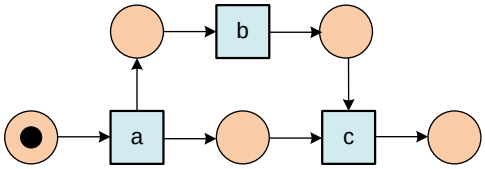


Fig. 12. Incorrect WF-net N_6 derived from $L_6 = [\langle a, c\rangle^2, \langle a, b, c\rangle^3, \langle a, b, b, c\rangle^2]$.

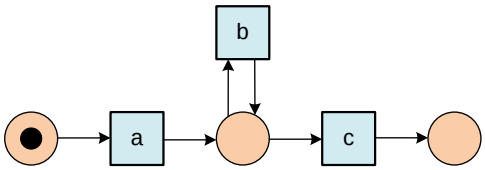


Fig. 13. WF-net N_7 having a so-called "short-loop".

A more difficult problem is the discovery of so-called non-local dependencies resulting from non-free choice process constructs. An example is shown in Figure 14. This net would be a good candidate after observing for example $L_8 = [\langle a, c, d\rangle^{45}, \langle b, c, e\rangle^{42}]$. However, the α-algorithm will derive the WF-net without the place labeled p_1 and p_2. Hence, $\alpha(L_8) = N_3$ shown in Figure 8 although the traces $\langle a, c, e\rangle$ and $\langle b, c, d\rangle$ do not appear in L_8. Such problems can be (partially) resolved using refined versions of the α-algorithm such as the one presented in [59].

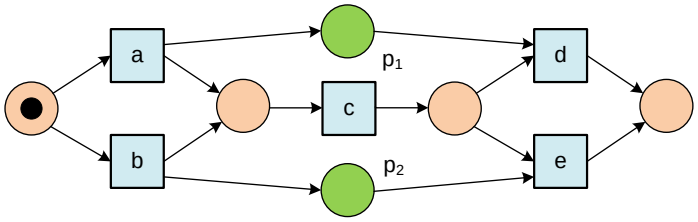


Fig. 14. WF-net N_8 having a non-local dependency.

The above examples show that the α-algorithm is able to discover a large class of models. The basic 8-line algorithm has some limitations when it comes to particular process patterns (e.g., short-loops and non-local dependencies). Some of these problems can be solved using various refinements. However, several more fundamental problems remain as shown next.

5 Challenges

The α-algorithm was one of the first process discovery algorithms 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 algorithms that use a completely different approach, e.g., genetic mining or synthesis based on regions [34]. Later we will describe some of these approaches. However, first we discuss the main requirements for a good process discovery algorithm.

To discover a suitable process model it is assumed that the event log contains a *representative sample of behavior*. There are two related phenomena that may make an event log less representative for the process being studied:

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

Often we would like to abstract from noise when discovering a process. This does not mean that noise is not relevant. In fact, the goal of conformance checking is to identify exceptions and deviations. However, for process discovery it makes no sense to include noisy behavior in the model as this will clutter the diagram and has little predictive value. Whereas noise refers to the problem of having "too much data" (describing rare behavior), completeness refers to the problem of having "too little data". To illustrate the relevance of completeness, consider a process consisting of 10 activities that can be executed in parallel and a corresponding log that contains information about 10,000 cases. The total number of possible interleavings in the model with 10 concurrent activities is 10! = 3,628,800. Hence, it is impossible that each interleaving is present in the log as there are fewer cases (10,000) than potential traces (3,628,800). Even if there are 3,628,800 cases in the log, it is extremely unlikely that all possible variations are present. For the process in which 10 activities can be executed in parallel, a local notion of completeness can reduce the required number of observations dramatically. For example, for the α-algorithm only $10 \times (10 - 1) = 90$ rather than 3,628,800 different observations are needed to construct the model.

Completeness and noise refer to qualities of the event log and do not say much about the quality of the discovered model. Determining the quality of a process mining result is difficult and is characterized by many dimensions. As shown in Figure 15, we identify four main quality dimensions: *fitness*, *simplicity*, *precision*, and *generalization* [2, 4, 51].

A model with good *fitness* allows for the behavior seen in the event log. A model has a perfect fitness if all traces in the log can be replayed by the model from beginning to end. There are various ways of defining fitness. It can be defined at the case level, e.g., the fraction of traces in the log that can be fully replayed. It can also be defined at the event level, e.g., the fraction of events in the log that are indeed possible according to the model [2, 4, 51]. Note that we defined an event log to be a multi-set of traces rather than an ordinary set:

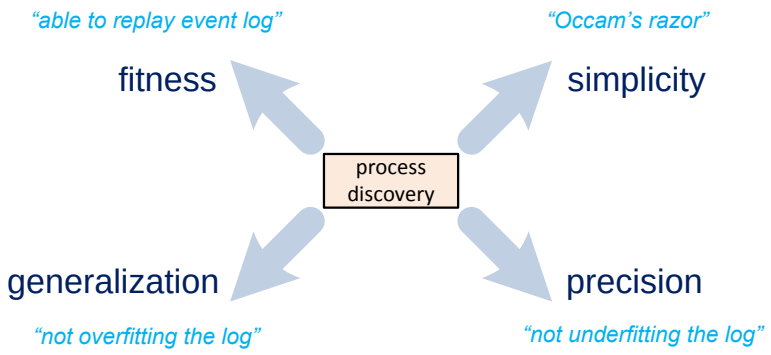


Fig. 15. Balancing the four quality dimensions: *fitness*, *simplicity*, *precision*, and *generalization* [2].

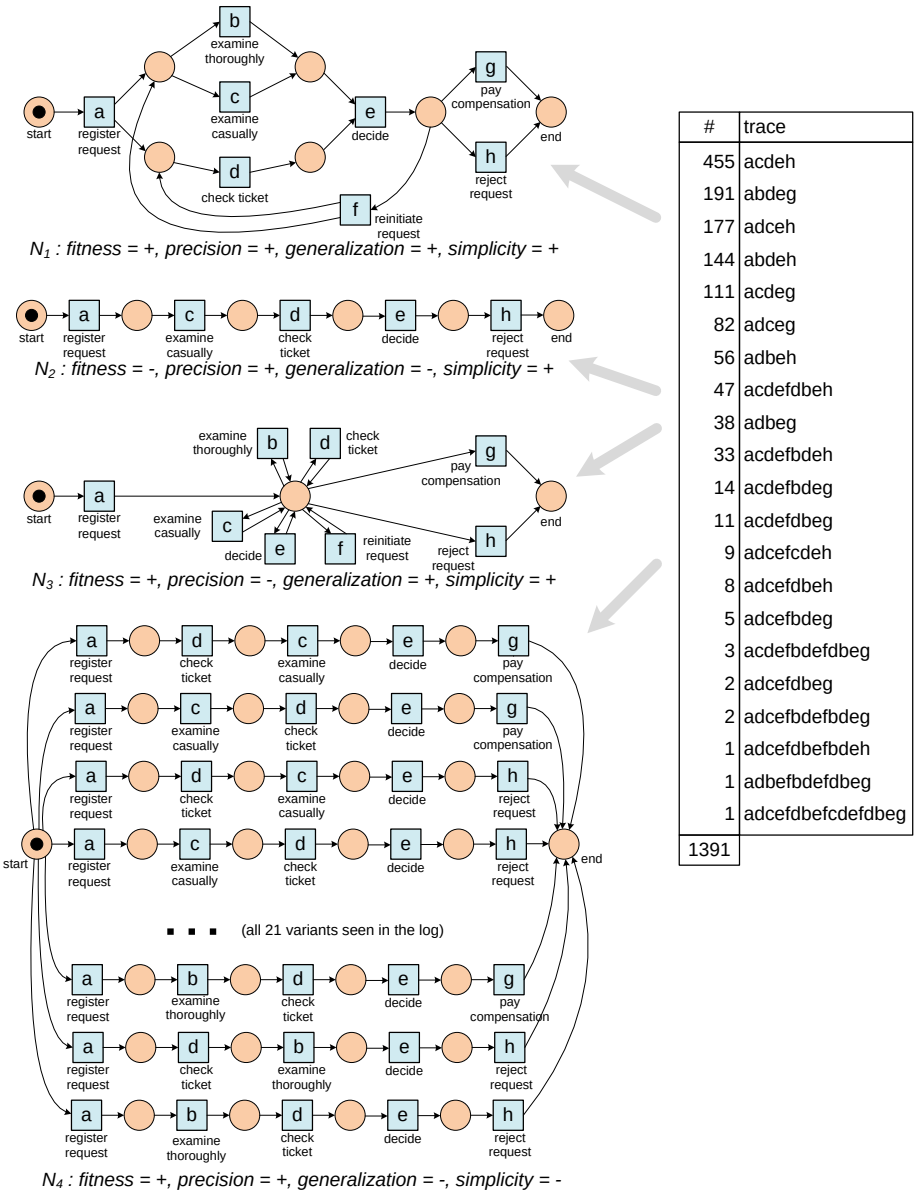
the frequencies of traces are important for determining fitness. If a trace cannot be replayed by the model, then the significance of this problem depends on the relative frequency.

The *simplicity* dimension in Figure 15 refers to *Occam's Razor*, the principle that states that "one should not increase, beyond what is necessary, the number of entities required to explain anything". Following this principle, we look for the "simplest process model" that can explain what is observed in the event log. The complexity of the model could be defined by the number of nodes and arcs in the underlying graph. Also more sophisticated metrics can be used, e.g., metrics that take the "structuredness" or "entropy" of the model into account.

Fitness and simplicity are obvious criteria. However, this is not sufficient as will be illustrated using Figure 16. Assume that the four models that are shown are discovered based on the event log also depicted in the figure. (Note that this event log was already shown in Section 1.) There are 1391 cases. Of these 1391 cases, 455 followed the trace $\langle a, c, d, e, h\rangle$. The second most frequent trace is $\langle a, b, d, e, g\rangle$ which was followed by 191 cases.

If we apply the α-algorithm to this event log, we obtain model N_1 shown in Figure 16. A comparison of the WF-net N_1 and the log shows that this model is quite good; it is simple and has a good fitness. WF-net N_2 models only the most frequent trace, i.e., it only allows for the sequence $\langle a, c, d, e, h\rangle$. Hence, none of the other $1391 - 455 = 936$ cases fits. WF-net N_2 is simple but has a poor fitness.

Let us now consider WF-net N_3, this is a variant of the so-called "flower model" [2, 51], i.e., a model that allows for all known activities at any point in time. Note that a Petri net without any places can replay any log and has a behavior similar to the "flower model" (but is not a WF-net). Figure 16 does not show a pure "flower model", but still allows for a diversity of behaviors. N_3 captures the start and end activities well. However, the model does not put any constraints on the other activities. For example trace $\langle a, b, b, b, b, b, b, f, f, f, f, f, g\rangle$



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

Fig. 16. Different Petri nets discovered for an event log containing 1391 cases.

is possible, whereas it seems unlikely that this trace is possible when looking at the event log, i.e., the behavior is very different from any of the traces in the log.

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

WF-net N_4 in Figure 16 reveals another potential problem. This model simply enumerates the 21 different traces seen in the event log. Note that N_4 is a so-called *labeled* Petri net, i.e., multiple transitions can have the same label (there are 21 transition with label a). The WF-net in Figure 16 is precise and has a good fitness. However, N_4 is also overly complex and is "overfitting". WF-net N_4 illustrates the need to *generalize*; one should not restrict behavior to the traces seen in the log as these are just examples. Overfitting is the problem that a very specific model is generated whereas it is obvious that the log only holds example behavior, i.e., the model explains the particular sample log, but a next sample log of the same process may produce a completely different process model. Recall that logs are typically far from complete. Moreover, generalization can be used to simplify models. WF-net N_1 shown in Figure 16 allows for behavior not seen in the log, e.g., $\langle a, d, c, e, f, d, b, e, f, c, d, e, h\rangle$. Any WF-net that restricts the behavior to only seen cases will be much more complex and exclude behavior which seems likely based on similar traces in the event log.

For real-life event logs it is challenging to balance the four quality dimensions shown in Figure 15. For instance, an oversimplified model is likely to have a low fitness or lack of precision. Moreover, there is an obvious trade-off between underfitting and overfitting [2, 4, 48, 51].

6 Process Discovery and the Theory of Regions

Problems similar to process discovery arise in other areas ranging from hardware design and to controller synthesis of manufacturing systems. Often the so called *theory of regions* is used to construct a Petri net from a behavioral specification (e.g., a language or a state space), such that the behavior of this net corresponds to the specified behavior (if such a net exists). The general question answered by the theory of regions is: *Given the specified behavior of a system, what is the Petri net that represents this behavior?*.

Two main types of region theory can be distinguished, namely *state-based region theory* and *language-based region theory*. The state-based theory of regions focusses on the synthesis of Petri nets from state-based models, where the state space of the Petri net is bisimilar to the given state-based model. The language-based region theory, considers a language over a finite alphabet as a behavioral specification. Using the notion of regions, a Petri net is constructed, such that all words in the language are firing sequences in that Petri net.

The aim of the theory of regions is to synthesize a precise model, with minimal generalization, while keeping a maximal fitness. The classical approaches described in this section (i.e., conventional *state-based* region theory and *language based* region theory) do not put much emphasis on simplicity. Unlike algorithms such as the heuristic miner [58], the genetic miner [47], and the fuzzy miner [39], conventional region-based methods do not compromise on precision in favor of simplicity or generalization.

In the remainder of this section, we introduce the main region theory concepts and discuss the differences between synthesis and process discovery. In Section 7 and Section 8 we show how region theory can be used in the context of process discovery.

6.1 State Based Region Theory

The *state-based region theory* [13, 15, 23, 24, 26, 27, 35] uses a transition system as input, i.e., it attempts to construct a Petri net that is bisimilar to the transition system. Hence both are behaviorally equivalent and if the system exhibits concurrency, the Petri net may be much smaller than the transition system.

Definition 9 (Transition system). *$TS = (S, E, T)$ defines a* labeled transition system *where S is the set of states, A is the set of visible activities (i.e., activities recorded in event log), $\tau \notin A$ is used to represent silent steps (i.e., actions not recorded in event log), $E = A \cup \{\tau\}$ is the set of transition labels, and $T \subseteq S \times E \times S$ is the transition relation. We use $s_1 \xrightarrow{e} s_2$ to denote a transition from state s_1 to s_2 labeled with e. Furthermore, we say that $S^s = \{s \in S \mid \nexists_{s' \in S, e \in E}\ s' \xrightarrow{e} s\} \subseteq S$ is the set of initial states, and $S^e = \{s \in S \mid \nexists_{s' \in S, e \in E}\ s \xrightarrow{e} s'\} \subseteq S$ is the set of final states.*

In the transition system, a *region* corresponds to a set of states such that all states have similarly labeled input and output edges. Figure 17 shows an example of a transition system. In fact, this figure depicts the reachability graph of the Petri net in Figure 5, where the states are anonymous, i.e., they do not contain information about how many tokens are in a place.

Definition 10 (State region). *Let $TS = (S, E, T)$ be a transition system and $S' \subseteq S$ a set of states. We say S' is a region, if and only if for all $e \in E$ one of the following conditions holds:*

1. *all the transitions $s_1 \xrightarrow{e} s_2$* enter *S', i.e., $s_1 \notin S'$ and $s_2 \in S'$,*
2. *all the transitions $s_1 \xrightarrow{e} s_2$* exit *S', i.e., $s_1 \in S'$ and $s_2 \notin S'$,*
3. *all the transitions $s_1 \xrightarrow{e} s_2$* do not cross *S', i.e., $s_1, s_2 \in S'$ or $s_1, s_2 \notin S'$*

Any transition system $TS = (S, E, T)$ has two trivial regions: $\emptyset$ (the empty region) and S (the region consisting of all states). Typically, only non-trivial regions are considered. A region r' is said to be a *subregion* of another region r if $r' \subset r$. A region r is *minimal* if there is no other region r' which is a subregion of r. Region r is a *preregion* of e if there is a transition labeled with e which

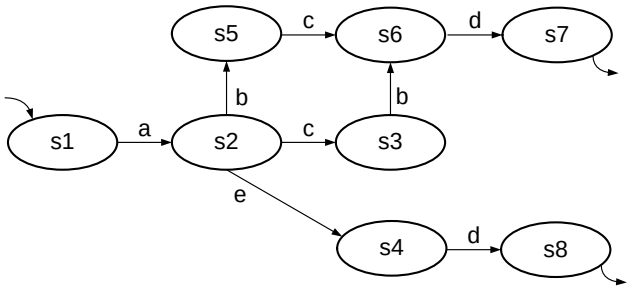


Fig. 17. A transition system with 8 states, 5 labels, 1 initial state and 2 final states.

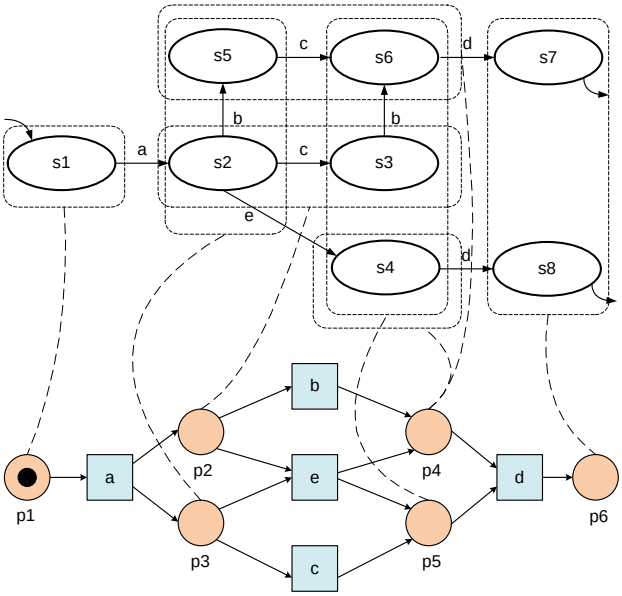


Fig. 18. The transition system of Figure 17 is converted into a Petri net using the "state regions". The six regions correspond to places in the Petri net.

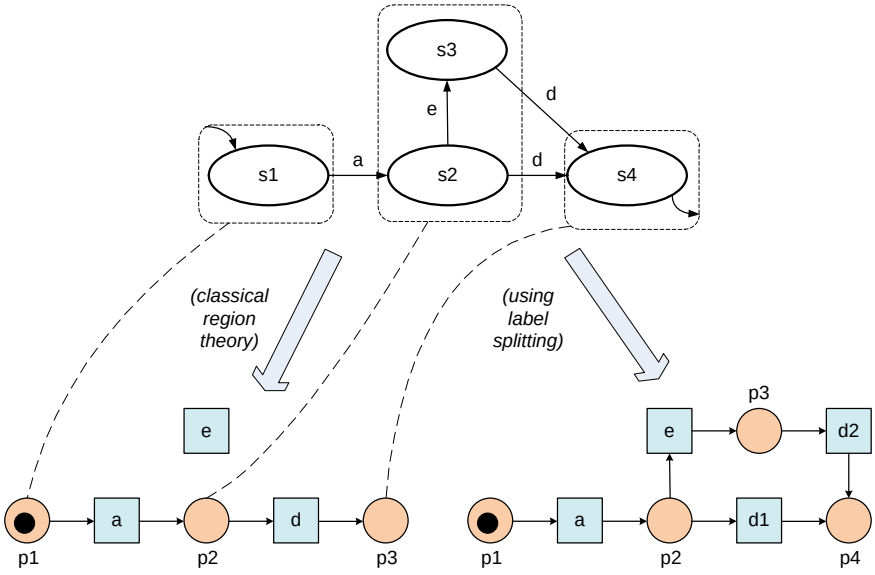


Fig. 19. The transition system is not elementary. Therefore, the generated Petri net using classical region theory is not equivalent (modulo bisimilarity). However, using "label-splitting" an equivalent Petri net can be obtained.

exits r. Region r is a *postregion* of e if there is a transition labeled with e which enters r.

For Petri net synthesis, a region corresponds to a Petri net *place* and an event corresponds to a Petri net *transition*. Thus, the main idea of the *synthesis algorithm* is the following: for each event e in the transition system, a transition labeled with e is generated in the Petri net. For each minimal region r_i a place p_i is generated. The flow relation of the Petri net is built the following way: $e \in p_i{}^\bullet$ if r_i is a preregion of e and $e \in {}^\bullet p_i$ if r_i is a postregion of e. Figure 18 shows the minimal regions of the transition system of Figure 17 and the corresponding Petri net.

The first publications on the theory of regions only dealt with a special class of transition systems called *elementary transition systems*. See [13, 15, 30] for details. The class of elementary transition systems is very restricted. In practice, most of the time, people need to deal with arbitrary transition systems that only by coincidence fall into the class of elementary transition systems. In the papers of Cortadella et al. [26, 27], a method for handling arbitrary transition systems was presented. This approach uses *labeled Petri nets*, i.e., different transitions can refer to the same event. (WF-net N_4 in Figure 16 is an example of a labeled Petri net, e.g., there are 21 transitions labeled a.) For this approach it has been shown that the behavior (cf. reachability graph) of the synthesized Petri net is *bisimilar* to the initial transition system even if the transition system is non-elementary.

More recently, in [23,24], an approach was presented where the constructed Petri net is not necessarily safe, but bounded[1]. Again, the reachability graph of the synthesized Petri net is bisimilar to the given transition system.

To illustrate the problem of non-elementary transition systems, consider Figure 19. This transition system is not elementary. The problem is that there are two states $s2$ and $s3$ that are identical in terms of regions, i.e., there is no region such that one is part of it and the other is not. As a result, the constructed Petri net on the left hand side of Figure 19 fails to construct a bisimilar Petri net. However, using label-splitting as presented in [26, 27], the Petri net on the right hand side can be obtained. This Petri net has two transitions $d1$ and $d2$ corresponding to activity d in the log. The splitting is based on the so-called notions of *excitation* and *generalized excitation region*, see [26]. As shown in [26, 27] it is always possible to construct an equivalent Petri net. However, label-splitting may lead to larger Petri nets. In [21] the authors show how to obtain the most precise model when label splitting is not allowed.

In state-based region theory, the aim is to construct a Petri net, such that its behavior is bisimilar to the given transition system. In process discovery however, we have a log as input, i.e., we have information about sequences of transitions, but not about states. In Section 7, we show how we can identify state information from event logs and then use state-based region theory for process discovery. However, we first introduce language-based region theory.

6.2 Language Based Region Theory

In addition to state-based region theory, we also consider language-based region theory [14, 17, 19, 28, 42, 43]. In their survey paper [45], Mauser and Lorenz show how for different classes of languages (step languages, regular languages and (infinite) partial languages) a Petri net can be derived such that the resulting net is the Petri net with the smallest behavior in which the words in the language are possible firing sequences.

Given a prefix-closed language $\mathcal{A}$ over some non-empty, finite set of activities A, the language-based theory of regions tries to find a finite Petri net $N(\mathcal{A})$ in which the transitions correspond to the elements in the set A and of which all sequences in the language are firing sequences (fitness criterion). Furthermore, the Petri net should minimize the number of firing sequences not in the language (precision criterion).

The Petri net $N(\mathcal{A}) = (\emptyset, A, \emptyset)$ is a finite Petri net with infinitely many firing sequences allowing for any sequence involving activities A. Such a model is typically underfitting, i.e., allowing for more behavior than suggested by the event log. Therefore, the behavior of this Petri net needs to be reduced so that the Petri net still allows to reproduce all sequences in the language, but does not allow for behavior unrelated to the examples seen in the event log. This is achieved by adding places to the Petri net. The theory of regions provides a method to identify these places, using *language regions*.

[1] A Petri net is safe if there can never be more than 1 token in any place. Boundedness implies that there exists an upper bound for the number of tokens in any place.

Definition 11 (Language Region). *Let A be a set of activities. A region of a prefix-closed language $\mathcal{L}$ over A is a triple $(\vec{x}, \vec{y}, c)$ with $\vec{x}, \vec{y} \in \{0,1\}^A$ and $c \in \{0,1\}$, such that for each non-empty sequence $w = w' \circ a \in \mathcal{L}$, $w' \in \mathcal{L}$, $a \in A$:*

$$c + \sum_{t \in A} \left(\vec{w'}(t) \cdot \vec{x}(t) - \vec{w}(t) \cdot \vec{y}(t) \right) \geq 0$$

This can be rewritten into the inequation system:

$$c \cdot \vec{1} + M' \cdot \vec{x} - M \cdot \vec{y} \geq \vec{0}$$

where M and M' are two $|\mathcal{L}| \times |A|$ matrices with $M(w,t) = \vec{w}(t)$, and $M'(w,t) = \vec{w'}(t)$, with $w = w' \circ a$. The set of all regions of a language is denoted by $\Re(\mathcal{L})$ and the region $(\vec{0}, \vec{0}, 0)$ is called the trivial region.[2]

Consider a region $r = (\vec{x}, \vec{y}, c)$ corresponding to some place p_r. For any prefix $w = w' \circ a$ in $\mathcal{L}$, region r satisfies $c + \sum_{t \in A} \left(\vec{w'}(t) \cdot \vec{x}(t) - \vec{w}(t) \cdot \vec{y}(t) \right) \geq 0$ where c is the initial number of tokens in place p_r, $\sum_{t \in A} \vec{w'}(t) \cdot \vec{x}(t)$ is the number of tokens produced for place p_r just before firing a (note that w' is the prefix without including the last a), and $\sum_{t \in A} \vec{w}(t) \cdot \vec{y}(t)$ is the number of tokens consumed from place p_r after firing a (w is the concatenation of w' and a). $\vec{w}$ is the Parikh vector of w, i.e., $\vec{w}(t)$ is the number of times t appears in sequence w. $\vec{x}(t)$ is the number of tokens t produces for place p_r. Transition t consumes $\vec{y}(t)$ tokens from place p_r per firing. So, $\vec{w}(t) \cdot \vec{y}(t)$ is the total number of tokens t consumes from place p_r when executing w.

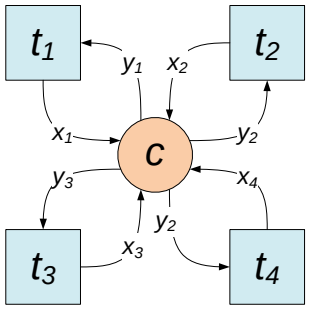


Fig. 20. Region for a language with four letters: t_1, t_2, t_3, and t_4.

Figure 20 illustrates the language-based region concept using for a language over four activities ($|A|$ =4), i.e., each solution $(\vec{x}, \vec{y}, c)$ of the inequation system can be regarded in the context of a Petri net, where the region corresponds to a

[2] To reduce calculation time, the inequation system can be rewritten to the form $[\vec{1}; M'; -M] \cdot \vec{r} \geq \vec{0}$ which can be simplified by eliminating duplicate rows.

feasible place with preset $\{t \mid t \in T, \vec{x}(t) \geq 1\}$ and postset $\{t \mid t \in T, \vec{y}(t) \geq 1\}$, and initially marked with c tokens. In this paper, we assume arc-weights to be 0 or 1 as we aim at understandable models (i.e., $\vec{x}, \vec{y} \in \{0,1\}^A$). As shown in [14, 16, 28, 43] it is possible to generalize the above notions to arbitrary arc-weights.

A place represented by a region can be added to a Petri net, without limiting its behavior with respect to traces seen in the event log. Therefore, we call such a place *feasible*.

Definition 12 (Feasible place). *Let $\mathcal{L}$ be a prefix-closed language over A and let $N = ((P, T, F), M)$ be a marked Petri net with $T = A$ and $M \in \mathbb{B}(P)$. A place $p \in P$ is called* feasible *if and only if there exists a* corresponding *region $(\vec{x}, \vec{y}, c) \in \Re(\mathcal{L})$ such that $M(p) = c$, and $\vec{x}(t) = 1$ if and only if $t \in {}^\bullet p$, and $\vec{y}(t) = 1$ if and only if $t \in p^\bullet$.*

In [16, 43] it was shown that any solution of the inequation system of Definition 11 can be added to a Petri net without influencing the ability of that Petri net to replay the log. However, since there are infinitely many solutions of that inequation system (assuming arc weights), there are infinite many feasible places and the authors of [16, 43] present two ways of finitely representing these places, namely a *basis representation* [43] and a *separating representation* [16, 43].

6.3 Process Discovery vs. Region Theory

When comparing region theory—state-based or language based—with process discovery, some important differences should be noted. First of all, in region theory, the starting point is a *full behavioral specification*, either in the form of a (possibly infinite) transition system, or a (possibly infinite) language. Hence, the underlying assumption is that the input is *complete* and *noise free* and therefore *maximal fitness* is assured.

Second, the aim of region theory is to provide a *compact, exact representation* of that behavior in the form of a Petri net. If the net allows for more behavior than specified, then this additional behavior can be proven to be minimal, hence region theory provides *precise* results.

Finally, when region theory is directly applied in the context of process discovery [16, 21, 53], the resulting Petri nets typically perform poorly with respect to two of the four dimensions shown in Figure 15. The resulting models are typically overfitting (lack of generalization) and are too difficult to comprehend (simplicity criterion). Therefore, in sections 7 and 8, we show how region theory can be modified for process discovery. The key idea is to allow the algorithms to generalize and relax on preciseness, with the aim of obtaining simpler models.

7 Process Discovery Using State-Based Region Theory

In Section 2 we introduced the concept of control-flow discovery and discussed the problems of existing approaches. In Section 6, we introduced region theory

and showed the main differences with control flow discovery. In this section, we introduce a two-step approach to combine process discovery with state-based region theory [8]. In the remainder, we elaborate on these two steps and discuss challenges.

7.1 From Event Logs to Transition Systems

In the first step, we construct a transition system from the log, where we generalize from the observed behavior. Furthermore, we "massage" the output, such that the region theory used in the second step is more likely to produce a simple model. In the second step, we use classical state-based region theory to obtain a Petri net. This section describes the first and most important step. Depending on the desired properties of the model and the characteristics of the log, the algorithm can be tuned to provide a more simple and/or generic model.

The most important aspect of process discovery is *deducing the states of the operational process in the log*. Most mining algorithms have an implicit notion of state, i.e., activities are glued together in some process modeling language based on an analysis of the log and the resulting model has a behavior that can be represented as a transition system. In this section, we propose to *define states explicitly* and start with the definition of a transition system.

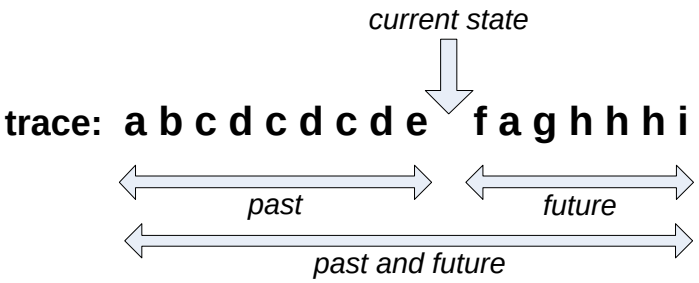


Fig. 21. Three basic "ingredients" can be considered as a basis for calculating the "process state": (1) past, (2) future, and (3) past and future.

In some cases, the state can be derived directly, e.g., each event encodes the complete state by providing values for all relevant data attributes. However, in the event log we typically only see activities and not states. Hence, we need to deduce the state information from the activities executed before and after a given state. Based on this, there are basically three approaches to defining the state of a partially executed case in a log:

- *past*, i.e., the state is constructed based on the history of a case,
- *future*, i.e., the state of a case is based on its future, or
- *past and future*, i.e., a combination of the previous two.

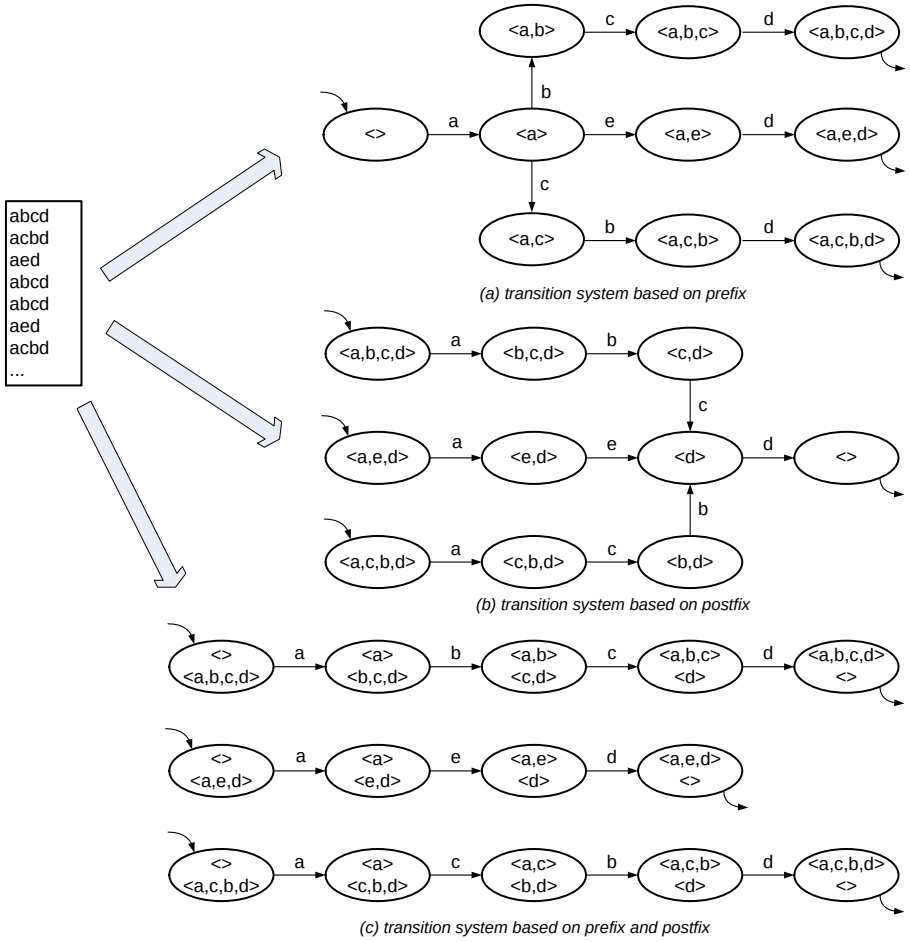


Fig. 22. Three transition systems derived from the log $L_1 = [\langle a, b, c, d\rangle^5, \langle a, c, b, d\rangle^8, \langle a, e, d\rangle^9]$.

Figure 21 shows an *example* of a trace and the three different "ingredients" that can be used to calculate state information. Given a concrete trace, i.e., the execution of a case from beginning to end, we can look at the state after executing the first nine activities. This state can be represented by the prefix, the postfix, or both.

To explain the basic idea of constructing a transition system from an event log, consider Figure 22. If we just consider the prefix (i.e., the past), we get the transition system shown in Figure 22(a). Note that the initial state is denoted $\langle\rangle$, i.e., the empty sequence. Starting from this initial state the first activity is always a in each of the traces. Hence, there is one outgoing arc labeled a, and the subsequent state is labeled $\langle a\rangle$. From this state, three transitions are possible

all resulting in different states, e.g., executing activity b results in state $\langle a, b\rangle$, etc. Note that in Figure 22(a) there is one initial state and three final states. Figure 22(b) shows the transition system based on postfixes. Here the state of a case is determined by its future. This future is known because process mining looks at the event log containing completed cases. Now there are three initial states and one final state. Initial state $\langle a, e, d\rangle$ indicates that the next activity will be a, followed by e and d. Note that the final state has label $\langle\rangle$ indicating that no activities need to be executed. Figure 22(c) shows a transition system based on both past and future. The node with label "$\langle a, b\rangle$,$\langle c, d\rangle$" denotes the state where a and b have happened and c and d still need to occur. Note that now there are three initial states and three final states.

The past of a case is a prefix of the complete trace. Similarly, the future of a case is a postfix of the complete trace. This may be taken into account completely, which leads to many different states and process models that may be too specific (i.e., "overfitting" models). It is also possible to take less information into account (e.g., just the last step in the process). This may result in "underfitting" models. The challenge is to select an abstraction that balances between "overfitting" and "underfitting". Many *abstractions* are possible; see for example the systematic treatment of abstractions in [8]. Here, we only highlight some of them.

Maximal horizon (h) The basis of the state calculation can be the complete prefix (postfix) or a *partial* prefix (postfix).

Filter (F) The second abstraction is to filter the (partial) prefix and/or postfix, i.e., activities in $F \subseteq A$ are kept while activities $A \setminus F$ are removed.

Maximum number of filtered events (m) The sequence resulting after filtering may contain a variable number of elements. Again one can determine a kind of horizon for this filtered sequence.

Sequence, bag, or set (q) The first three abstractions yield a sequence. The fourth abstraction mechanism optionally removes the order or frequency from the resulting trace.

Visible activities (V) The fifth abstraction is concerned with the transition labels. Activities in $V \subseteq A$ are shown explicitly on the arcs while the activities in $A \setminus V$ are not shown.

Figure 23 illustrates the abstractions. In Figure 23(a) only the set abstraction is used $q = set$. The result is that several states are merged (compare with Figure 22(a)). In Figure 23(b) activities b and c are filtered out (i.e., $F = \{a, d, e\}$ and $V = \{a, d, e\}$). Moreover, only the last non-filtered event is considered for constructing the state (i.e., $m = 1$). Note that the states in Figure 23(b) refer to the last event in $\{a, d, e\}$. Therefore, there are four states: $\langle a\rangle$, $\langle d\rangle$, $\langle e\rangle$, and $\langle\rangle$. It is interesting to consider the role of b and c. First of all, they are not considered for building the state ($F = \{a, d, e\}$). Second, they are also not visualized ($V = \{a, d, e\}$), i.e., the labels are suppressed. The corresponding transitions are collapsed into the unlabeled arc from $\langle a\rangle$ to $\langle a\rangle$. If V would have included b and c, there would have been two such arcs labeled b respectively c.

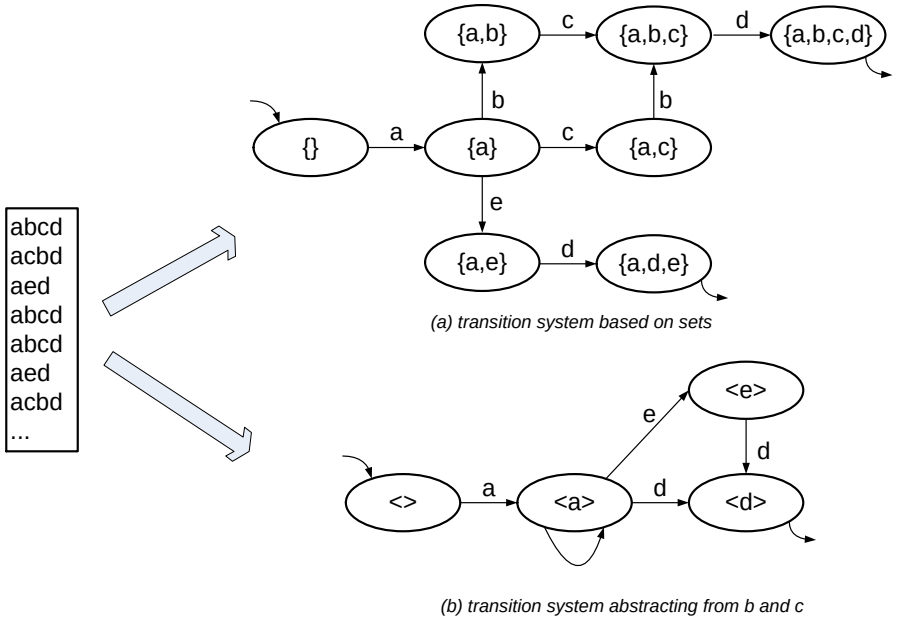


Fig. 23. Two transition systems built on L_1 using the following prefix abstractions: (a) $h = \infty$, $F = A$ (i.e., all activities), $m = \infty$, $q = set$, and $V = A$, and (b) $h = \infty$, $F = \{a, d, e\}$, $m = 1$, $q = seq$, and $V = \{a, d, e\}$.

The first four abstractions can be applied to the prefix, the postfix, or both. In fact, different abstractions can be applied to the prefix and postfix. As a result of these choices many different transitions systems can be generated. If more rigorous abstractions are used, the number of states will be smaller and the danger of "underfitting" is present. If, on the other hand, fewer abstractions are used, the number of states may be larger resulting in an "overfitting" model. An extreme case of overfitting was shown in Figure 22(c). At first this may seem confusing; however, as indicated in the introduction it is important to provide a repertoire of process discovery approaches. Depending on the desired degree of generalization, suitable abstractions are selected and in this way the analyst can balance between overfitting and underfitting, i.e., between generalization and precision in a controlled way.

Using classical region theory, we can transform the transition system into a process model. However, while we can now balance precision and generalization, we did not focus on simplicity yet. Therefore, we make use of the inner workings of state-based region theory to "massage" the transition system. This is intended to "pave the path" for region theory. For example, one may remove all "self-loops", i.e., transitions of the form $s \xrightarrow{a} s$ (cf. Figure 24(a)). The reason may be that one is not interested in events that do not change the state or that the synthesis algorithm in the second step cannot handle this. Another example

would be to close all "diamonds" as shown in Figure 24(b). If $s_1 \xrightarrow{a_1} s_2$, $s_1 \xrightarrow{a_2} s_3$, and $s_2 \xrightarrow{a_2} s_4$, then $s_3 \xrightarrow{a_1} s_4$ is added. The reason for doing so may be that because (1) both a_1 and a_2 are enabled in s_1 and (2) after doing a_1, activity a_2 is still enabled, it is assumed that a_1 and a_2 can be executed in parallel. Although the sequence $\langle a_2, a_1 \rangle$ was not observed, it is assumed that this is possible and hence the transition system is extended by adding $s_3 \xrightarrow{a_1} s_4$.

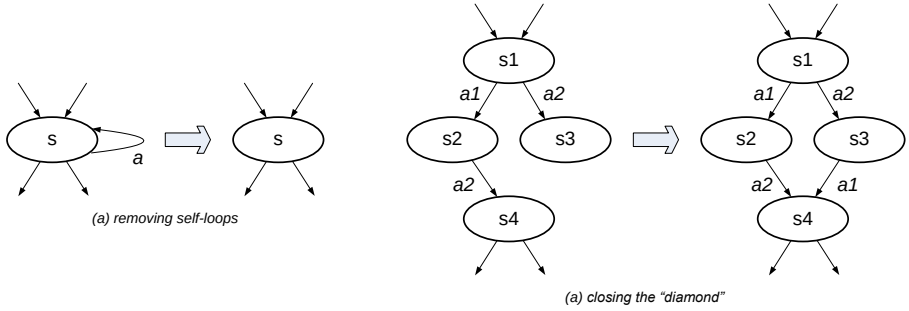


Fig. 24. Two examples of modifications of the transition system to aid the construction of the process model.

7.2 From Transition Systems to Petri Nets

In the second step, the transition system is transformed into a Petri net using the techniques described in [13,15,23,24,26,27,35]. In Section 6.1, we introduced the basic idea of state-based regions. Therefore, we do not elaborate on this here. The important thing to note is that there is range of techniques to convert a transition system into a Petri net. These techniques typically only address two of the four quality dimensions mentioned in Figure 15: *fitness* and *precision*. The other two dimensions—*simplicity* and *generalization*—need to be addressed when constructing the transition system or by imposing additional constraints on the Petri net.

The goal of process mining is to present a model that can be interpreted easily by process analysts and end-users. Therefore, complex patterns should be avoided. Region-based approaches have a tendency to introduce "smart places", i.e., places that compactly serve multiple purposes. Such places have many connections and may have non-local effects (i.e., the same place is used for different purposes in different phases of the process). Therefore, it may be useful to guide the generation of places such that they are easier to understand. This is fairly straightforward in both state-based region theory and language-based region theory. In [26, 27] it is shown that additional requirements can be added with respect to the properties of the resulting net. For example, the net can be forced to be free-choice, pure, etc. See [8] for examples.

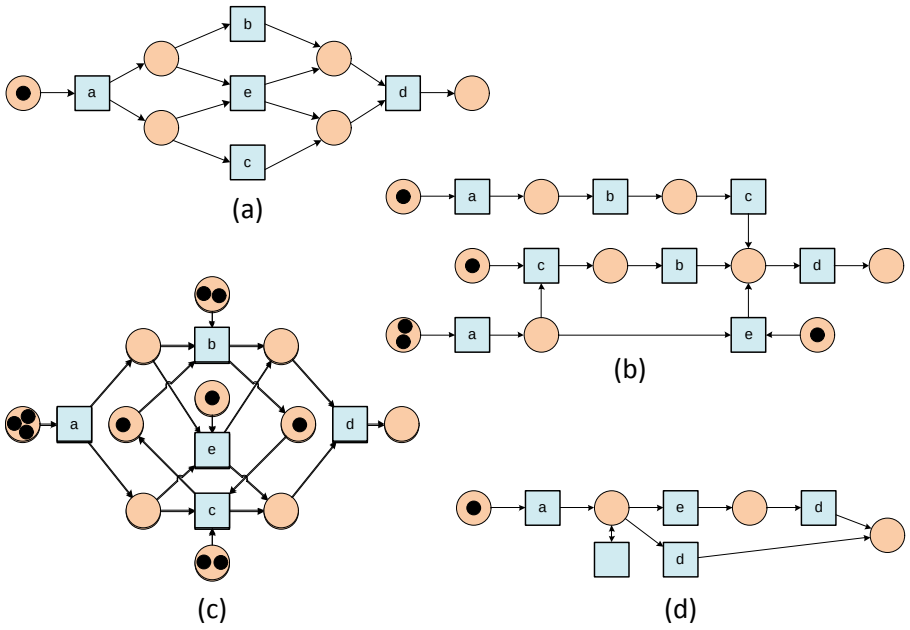


Fig. 25. Various Petri nets derived for the transitions systems in figures 22 and 23 using state-based regions. All models are based on event log $L_1 = [\langle a, b, c, d\rangle^5, \langle a, c, b, d\rangle^8, \langle a, e, d\rangle^9]$.

The approach was already illustrated using Figure 18. Figure 25 shows some more examples based on the transition systems in figures 22 and 23. These models where computed using the classical synthesis approach presented in [26, 27]. This approach applies label-splitting if needed. Note that all transition systems were derived from event log $L_1 = [\langle a, b, c, d\rangle^5, \langle a, c, b, d\rangle^8, \langle a, e, d\rangle^9]$. The Petri net in Figure 25(a) is obtained by applying state-based region theory to the transition system in Figure 22(a). The same model is obtained when computing the regions for the transition system in Figure 23(a). The Petri net in Figure 25(b) is obtained when applying state-based region theory to the transition system in Figure 22(b). Two things can be noted: (1) the multiple initial states in Figure 22(b) result in many initial tokens and source places, and (2) label splitting is used (e.g., there are two a transitions) to allow for the multiple starting points. The region-based approach synthesizes the model in Figure 25(c) for the transition system in Figure 22(c). Also this model suffers from the problem that there are multiple initial states. In general, we suggest to avoid having multiple initial states in the transition system to be synthesized. It is trivial to merge the initial states or add a new artificial initial state before applying region-based synthesis. Figure 25(d) was obtained from the transition system in Figure 23(b). The Petri net shows that if we abstract from b and c, we obtain an unlabeled

transition indicating the state in which b and c would have occurred. This silent transition is due to the self-loop in the transition system of Figure 23(b). Eliminating the self-loop using the strategy presented in Figure 24(a) would remove the unlabeled transition in Figure 25(d).

7.3 Challenges

In this section, we have shown that by combining abstraction techniques and region theory, a powerful process mining algorithm can be obtained. Through several abstractions, we can obtain the desired level of precision and generalization, while by massaging the transition system, we can try to obtain simple models. However, there are also some drawbacks of this approach.

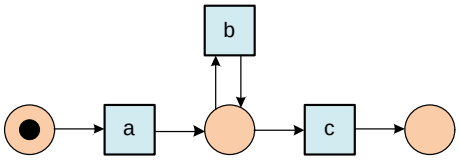


Fig. 26. Petri net obtained using region theory applied to log $L_6 = [\langle a, c\rangle^2, \langle a, b, c\rangle^3, \langle a, b, b, c\rangle^2]$ using the following settings: $h = \infty$, $F = A$ (i.e., all activities), $m = \infty$, $q = set$, and $V = A$ and a post-processing step in which states with identical inflow or outflow are merged.

It is far from trivial to select the "right" parameters for the abstractions. Existing techniques and tools are sensitive to changes of parameter values, and the result is often unpredictable. Hence, obtaining a suitable process model is a matter of trial-and-error. Figure 26 shows, for example, the settings with which we can obtain the desired model for log L_6, i.e., the Petri net with a self-loop on transition b. However, the model shown in Figure 27 illustrates that the wrong settings may lead to an overfitting model.

Nonetheless, the state-based region approach is one of the few that can detect long-term dependencies, as shown by Figure 28, which resulted from applying the technique to log L_8.

Furthermore, the major drawback of the approach outlined here is the computational complexity. For larger logs, the resulting transition system may not fit in main memory and second, the region theory used to obtain a Petri net has a time complexity which is exponential in the size of the transition system.

8 Process Discovery Using Integer Linear Programming

In Section 7, we have presented a two-step approach to apply region theory in the context of process mining. We focussed on obtaining a transition system

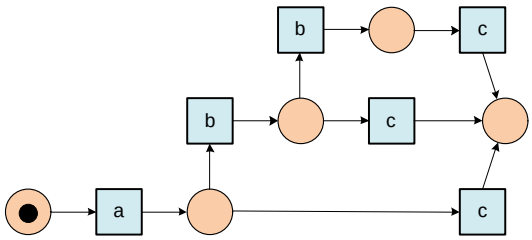


Fig. 27. Petri net obtained using region theory applied to log L_6 using the following parameters: $h = \infty$, $F = A$ (i.e., all activities), $m = \infty$, $q = multiset$, and $V = A$.

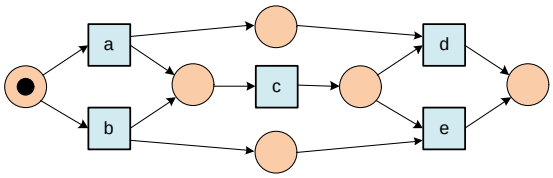


Fig. 28. Petri net obtained using region theory applied to log $L_8 = [\langle a, c, d\rangle^{45}, \langle b, c, e\rangle^{42}]$ using the following settings: $h = \infty$, $F = A$ (i.e., all activities), $m = \infty$, $q = set$, and $V = A$.

from an event log and used classical region theory to obtain a Petri net. In this section, we do not consider the region theory as a black box, but instead, we extend existing approaches to make them more applicable in the context of process discovery, mainly by allowing the techniques to generalize from the log and to produce simpler models.

Both a basis and the separating representations of regions presented in [16,43] are based on the same principle, namely that a finite representation is provided of the infinite set of places satisfying Definition 11. By doing so, the language-based region theory ensures maximal preciseness and fitness, with little to no generalization and no aim for simple models. Hence, only two of the four quality dimensions of Figure 15 are considered.

For process discovery, we are aiming at simple, generalizing models. Hence, we present an approach [60], where we only represent those places satisfying Definition 11 that:

- each place expresses a causal dependency clearly visible in the log,
- no implicit places are included in the net, and
- places which are more expressive than others are favored, i.e., places with minimal input transitions and maximal output transitions are favored.

In contrast to the state-based region approach, we do not try to influence generalization and precision directly. Instead, we focus on model simplicity, by

limiting the number of places in the model (and allowing for varying this number). As with the state-based approach, maximal fitness is guaranteed. In order to select places satisfying Definition 11, we convert this equation into a Integer Linear Programming (ILP) problem.

8.1 Integer Linear Programming Representation

We quantify the expressiveness of places, in order to provide a target function, necessary to translate the inequation system of Definition 11 into an *Integer Linear Programming* (ILP) problem. In Section 8.2, we then use the result to generate a Petri net in a step-by-step fashion. In section 8.3, we provide insights into the causal dependencies found in a log and how these can be used for finding places.

To apply the language-based theory of regions in the field of process discovery, we need to represent the event log as a prefix-closed language, i.e., by all the traces present in the event log, and their prefixes. Recall from Definition 1 that an event log is a finite bag of traces.

Definition 13 (Language of an event log). *Let A be a set of activities. Let $L \in \mathbb{B}(A^*)$ be an event log over this set of activities. The language $\mathcal{L}$ that represents this event log, uses alphabet A, and is defined by:*

$$\mathcal{L} = \{\sigma \in A^* \mid \exists \sigma' \in L \colon \sigma \leq \sigma'\}$$

A trivial Petri net capable of reproducing a language is a net with only transitions. This net is simple, can represent all traces, and hence has maximal fitness. It also generalizes well, but the Petri net with only transitions is very imprecise because anything is possible according to the model. To restrict the behavior allowed by the Petri net, but not observed in the log, we start adding places to that Petri net. As stated before, the places we add to the Petri net should be *as expressive as possible*, which is the same as saying that such places have a maximal postset and a minimal preset, i.e., it should not be possible to add an output transition to or to remove an input transition from a place without reducing the fitness of the resulting net.

Besides searching for regions that lead to places with maximum expressiveness, we also want to avoid adding implicit places to a model. Therefore, we will search for "minimal regions" as introduced in [30]. Using the inequation system of Definition 11 and the expressiveness of a place, we can define a target function for our ILP problem to construct the places of a Petri net in a logical order [52].

The following target function is shown to be such that it favors minimal regions which are maximally expressive [60]:

Definition 14 (Target function). *Let A be a set of activities. Let $L \in \mathbb{B}(A^*)$ be an event log and $\mathcal{L}$ the corresponding language. Furthermore, let M be the matrix defined in Definition 11. We define the function $\tau : \Re(\mathcal{L}) \to \mathbb{N}$ by*

$$\tau((\vec{x}, \vec{y}, c)) = c + \vec{1}^{\,T}(\vec{1} \cdot c + M \cdot (\vec{x} - \vec{y}))$$

Combining this target function with the inequation system of Definition 11 yields the following ILP problem:

Definition 15 (ILP formulation). *Let A be a set of activities, let $L \in \mathbb{B}(A^*)$ be an event log, and let M and M' be the matrices as defined in Definition 11. We define the ILP ILP_L for event log L as:*

$$
\begin{array}{lrll}
\textit{Min } c + \vec{1}^{\,T}(\vec{1} \cdot c + M \cdot (\vec{x} - \vec{y})) & & & \textit{Definition 14} \\
\textit{s.t.} & c \cdot \vec{1} + M' \cdot \vec{x} - M \cdot \vec{y} \geq \vec{0} & & \textit{Definition 11} \\
& \vec{1}^{\,T} \cdot \vec{x} + \vec{1}^{\,T} \cdot \vec{y} \geq 1 & & \textit{There should be at least one edge} \\
& \vec{0} \leq \vec{x} \leq \vec{1} & x \in \{0,1\}^{|T|} & \\
& \vec{0} \leq \vec{y} \leq \vec{1} & y \in \{0,1\}^{|T|} & \\
& 0 \leq c \leq 1 & c \in \{0,1\} &
\end{array}
$$

This ILP problem provides the basis for our process discovery problem. However, an optimal solution to this ILP only provides a single feasible place with a minimal value for the target function. Therefore, in the next section, we show how this ILP problem can be used as a basis for constructing a Petri net from a log.

8.2 Constructing Petri Nets Using ILP

In the previous subsection, we provided the basis for adding places to a Petri net based on knowledge extracted from a log. In fact, the target function of Definition 14 provides a partial order on all elements of the set $\Re(\mathcal{L})$, i.e., the set of all regions of a language. In this subsection, we show how to generate the first n places of a Petri net, that is (1) able to reproduce a log under consideration and (2) of which the places are as expressive as possible.

A trivial approach would be to add each found solution as a negative example to the ILP problem, i.e., explicitly forbidding this solution. However, it is clear that once a region r has been found and the corresponding feasible place is added to the Petri net, we are no longer interested in regions r' for which the corresponding feasible place has more tokens, less outgoing arcs or more incoming arcs, i.e., we are only interested in independent regions.

Definition 16 (Refining the ILP after each solution). *Let A be a set of activities, let $L \in \mathbb{B}(A^*)$ be an event log, let M and M' be the matrices as defined in Definition 11 and let ILP_L^0 be the corresponding ILP. Furthermore, for $i \geq 0$ let region $r_i = (\vec{x_i}, \vec{y_i}, c_i)$ be a minimal solution of ILP_L^i. We define the refined ILP as ILP_L^i, with the extra constraint specifying that:*

$$-c_i \cdot c + \vec{y}^{\,T} \cdot (\vec{1} - \vec{y_i}) - \vec{x}^{\,T} \cdot \vec{x_i} \geq -c_i + 1 - \vec{1}^{\,T} \cdot \vec{x_i}$$

Note that for any solution $r = (\vec{x}, \vec{y}, c)$ of ILP_L^i: $c < c_i$ or there is a $t \in A$ such that $\vec{x}(t) < \vec{x_i}(t)$ or $\vec{y}(t) > \vec{y_i}(t)$. If this is not the case (i.e., $c \geq c_i$ and $\vec{x}(t) \geq \vec{x_i}(t)$ and $\vec{y}(t) \leq \vec{y_i}(t)$ for any t), then $-c_i \cdot c = -c_i$, $-\vec{x}(t) \cdot \vec{x_i}(t) = -x_i(t)$, and $\vec{y}(t) \cdot \vec{y_i}(t) = 0 \not\geq 1$. Hence, we find a contradiction with respect to the extra

constraint. As a result the new region r is forced to be sufficiently different from r_i.

The refinement operator presented above, basically defines an algorithm for constructing the places of a Petri net that is capable of reproducing a given log. The places are generated in an order which ensures that the most expressive places are found first and that only places are added that have less tokens, less outgoing arcs, or more incoming arcs. Furthermore, each solution of a refined ILP is also a solution of the original ILP, since the new solution satisfies all constraints of the initial ILP formulation, and some extra constraints. Hence, all places constructed using this procedure are feasible places.

This procedure, can be used to continue adding places, thus making the model more precise, while compromising on model complexity as shown by Figure 29. The Petri net in Figure 29 allows for more behavior than the log L_1 contains, so in theory more places could still be added. Nonetheless, any new place would be such that it has fewer output arcs, or more input arcs than the ones included in this model. In the worst case, the total number of places introduced is exponential in the number of transitions. Since there is no way to provide insights into an upperbound for the number of places to generate, we propose a more suitable approach, *not using the refinement step of Definition 16*. Instead, we propose to guide the search for solutions (i.e. for places) using concepts from the α-algorithm [9].

8.3 Using Log-Based Properties

Recall from the beginning of this section, that we are specifically interested in places expressing explicit causal dependencies between transitions. In this subsection, we use the causal relations $\rightarrow_L$ defined earlier in Definition 7 in combination with the ILP of Definition 15 to construct a Petri net.

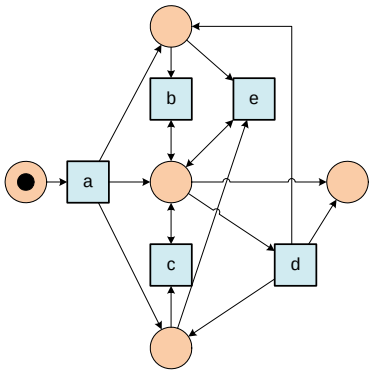


Fig. 29. Petri net obtained using language-based region theory naively applied to log L_1.

Causal dependencies between transitions are used by many process discovery algorithms [6,9,31,58] and generally provide a good indication as to which transitions should be connected through places. Furthermore, extensive techniques are available to derive causal dependencies between transitions using heuristic approaches [9, 31]. However, it is not known whether the log is complete and whether we covered all causal dependencies. Therefore, we restrict ourselves to search for a Petri net such that if a causal dependency is not in the log, it is also not in the net. In order to find a place expressing a specific causal dependency, we extend the ILP presented in Definition 15.

Definition 17 (ILP for causal dependency). *Let A be a set of activities, let $L \in \mathbb{B}(A^*)$ be an event log, let M and M' be the matrices as defined in Definition 11 and let ILP_L be the corresponding ILP. Furthermore, let $t_1, t_2 \in A$ and assume $t_1 \rightarrow_L t_2$. We define the refined ILP, $ILP_L^{t_1 \rightarrow t_2}$ as ILP_L, with two extra bounds specifying that:*

$$\vec{x}(t_1) = \vec{y}(t_2) = 1$$

A solution of the optimization problem expresses the causal dependency $t_1 \rightarrow_L t_2$, and restricts the behavior as much as possible. However, such a solution does not have to exist, i.e., the ILP might be infeasible, in which case no place is added to the Petri net being constructed. Nonetheless, by considering a separate ILP for each causal dependency in the log, a Petri net can be constructed, in which each place is as expressive as possible and expresses at least one dependency derived from the log. With this approach, at most one place is generated for each dependency and thus the upper bound of places in $N(\mathcal{L})$ is the number of causal dependencies, which is worst-case quadratic in the number of transitions.

The result of applying this log-based technique to our log L_1 is shown in Figure 30. This model is very close to the desired model, except that it does not contain a final place. This is a general drawback of language-based region theory: the focus is on the ability to reproduce prefixes of log traces rather than termination in a well-defined final state.

Up to now, we did not impose any restriction on the structure of the resulting Petri net. By adding constraints, several Petri net properties can be expressed, thus resulting in elementary nets, pure nets, (extended) free-choice nets, state machines and marked graphs [60]. This allows us to further simplify the resulting Petri net. Note that this is similar to the refinement described in Section 7.2 for state-based regions.

8.4 Challenges

In sections 7 and 8 we presented several ways to use region theory in the context of process discovery in order to alleviate some of the problems of the α-algorithm. First, we have shown how to we can balance precision and generalization while constructing a transition system from a log. Then, by massaging the transition

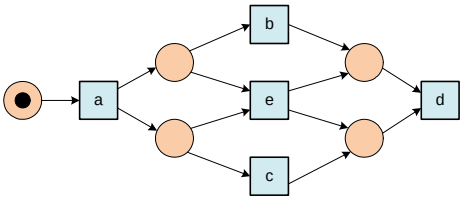


Fig. 30. Petri net obtained using language-based region theory using log-based properties applied to log L_1. Note that compared to earlier solutions the sink place denoting termination is missing.

system, we can somewhat improve the simplicity of the resulting models. When using language-based region theory, we have shown that we can focus on the simplicity of the resulting model. By incrementally introducing places, we can make the resulting model more precise in a step-by-step fashion. Figures 31 and 32 show that we can discover models for the logs L_6 and L_8, but the long-term dependency in L_8 is not identified, due to the reliance on the causal dependencies used in the α-algorithm. Furthermore, as discussed before, language-based regions have problems making the final state explicit (i.e., sink places are missing in figures 31 and 32).

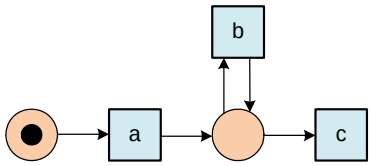


Fig. 31. Petri net discovered for event log L_6. The model was obtained using language-based region theory guided by log-based properties.

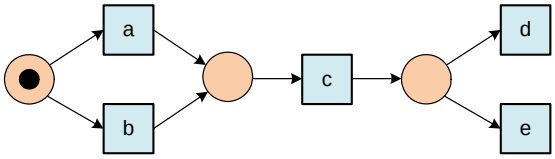


Fig. 32. Petri net obtained using language-based region theory guided by log-based properties applied to log $L_8 = [\langle a, c, d\rangle^{45}, \langle b, c, e\rangle^{42}]$. No sink place is created and the long-term dependencies are not discovered because only short-term dependencies are used to guide the discovery of places.

Unfortunately, all region-based approaches are computationally challenging. In the case of the language-based regions, finding a solution for each incremental ILP problem is of worst-case exponential time complexity. Furthermore, the common property of all region-based techniques is that the fitness of the discovered net is guaranteed to be 100%, regardless of the log. This makes these approaches very robust, but also sensitive to noise.

Thus far we only used toy examples to illustrate the different concepts. All functionality has been embedded in the process discovery framework ProM, which is capable of constructing nets for logs with thousands of cases referring to dozens of transitions. The techniques have been tested on many real-life and synthetic event logs. However, a discussion of these experimental results is outside the scope of this article. For this we refer to [2, 7, 39, 47, 53].

9 Tool Support

Both for process mining and region theory, it is essential that algorithms can be put to the test in real life environments. Therefore, almost all work presented in this article is implemented in freely available tools. For example, classical state-based region theory is implemented in Petrify and Genet [22], while Rbminer [54] applies this in a process discovery context. Some of the language-based region theory is implemented in VIPTool [18].

The process mining algorithms presented in sections 4, 7 and 8 have all been implemented in the *ProM framework* [11, 56, 57]. All algorithms discussed in this article can be found the most recent version of ProM (version 6.0 and later). ProM is a generic open-source framework for implementing process mining algorithms in a standard environment.

Figure 33 shows the startup screen of ProM. Here, a log was opened for analysis which is shown in the workspace. When selecting the log and clicking on the action button, the user is taken to the action browser, where in Figure 34, the α-miner is selected. The α-miner is an implementation of the work in Section 4.

In earlier versions of ProM, the actual process mining algorithms implemented by plug-ins assumed the presence of a GUI. Most algorithms require parameters, and the plug-in would ask the user for these parameters using some GUI-based dialog. Furthermore, some plug-ins displayed status information using progress bars and such. Thus, the actual process mining algorithm and the use of the GUI were intertwined. As a result, the algorithm could only be run in a GUI-aware context, say on a local workstation. This way, it was impossible to effectively run process mining experiments using a distributed infrastructure and/or in batch.

In ProM 6, the process mining algorithm and the GUI have been carefully separated, and the concepts of *contexts* has been introduced. For a plug-in, the context is the proxy for its environment, and the context determines what the plug-in can do in its environment. A plug-in can only display a dialog or a progress bar on the display if the context is GUI-aware. Typically, in ProM 6, the implementation of an algorithm is split into a number of plug-ins: A plug-in

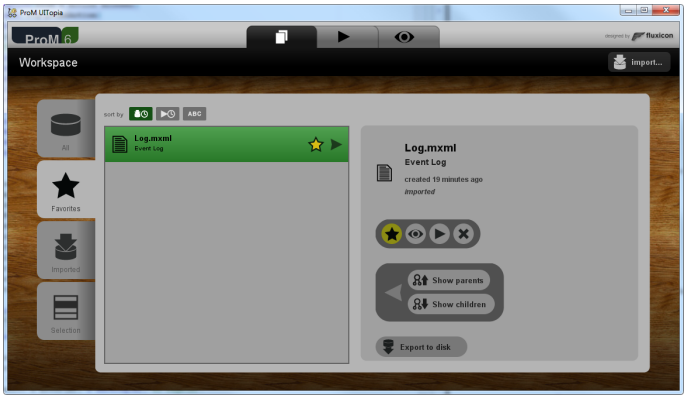


Fig. 33. ProM 6 Workspace; opening screen after loading a file.

Fig. 34. ProM 6 Action Browser; selecting the *alpha*-miner to discover a process model from the loaded event log.

Fig. 35. ProM 6 Package manager showing the packages relevant for the techniques presented in this article.

for every context. The actual process mining algorithm will be implemented in a generic way, such that it can run in a general (GUI-unaware) context. This allows the algorithm to be run in any context, even in a distributed context [20]. The dialog for setting the required parameters is typically implemented in a GUI-aware variant of the plug-in. Typically, this GUI-aware plug-in first displays the parameter dialog, and when the user has provided the parameters and has closed the dialog, it will simply run the generic plug-in using the provided parameters.

The major advantage of this is that the ProM framework may decide to have the generic plug-in run on a different computer than the local workstation. Some plug-ins may require lots of system resources (e.g., computing power, memory, and disk space), like for example the genetic miner. Basically, the genetic miner takes a model and a log, and then generates a number of alternative models for the given log. The best of these alternative models are then taken as new starting points for the genetic miner. The genetic miner repeats this until some stop criterion has been reached, after which it returns the best model found so far. Clearly, this miner might take considerable time (it may take hundreds of iterations before it stops and the fitness calculation is very time-consuming for large logs), and it may take considerable memory (the number of alternative models may grow rapidly). For such an algorithm, it might be preferable to have it run on a server which is more powerful than the local workstation. Moreover, genetic mining can be distributed in several ways [20]. For example, the pop-

ulation can be partitioned over various nodes. Each subpopulation on a node evolves independently for some time after which the nodes exchange individuals. Similarly, the event logs may be portioned over nodes thus speeding up the fitness calculations.

Besides separating the functionality from the user interface, ProM 6 requires functionality to be provided in *packages*. These packages each contain a collection of related algorithms, typically implemented by one research group. When ProM is started for the first time, the package manager is opened as shown in Figure 35. Here, for each known package, ProM shows who the author is, what the current version is and whether or not this version is installed. The work presented in this article, requires the following packages to be installed: `AlphaMiner`, `TransitionSystems` and `ILPMiner`. The other packages shown are automatically installed due to dependencies. Furthermore, the package `Petrify` provides import and export functionality to and from the state-based region tool Petrify.

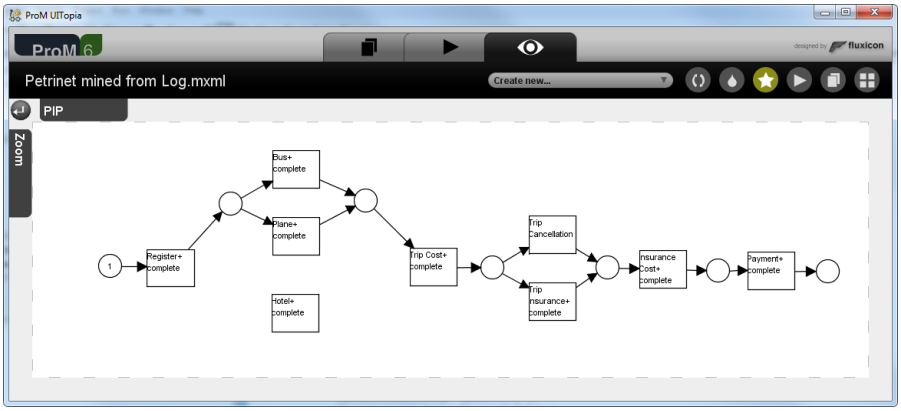


Fig. 36. Result of α-miner: the α-algorithm has problems dealing with the multiple hotel bookings interleaved with other booking activities.

The event log opened in Figure 33 is a log consisting of 1000 cases of a travel agency. A customer registers, then purchases a bus ticket or a plane ticket while at the same time he books one or more hotels. After the booking phase, the trip costs are computed and the customer has to choose between two types of insurance. After that, the total costs are calculated and the payment is completed. This is a rather simple example used to show the results of the three algorithms.

The resulting Petri net after applying the α-algorithm to this log is shown in Figure 36. The result after executing the transition system miner is shown in Figure 37 and the result of the ILP miner is shown in 38. All three algorithms provide a model that indeed models the given situation. The difficulty here is the fact that the hotel booking is executed one or more times. The α-algorithm does

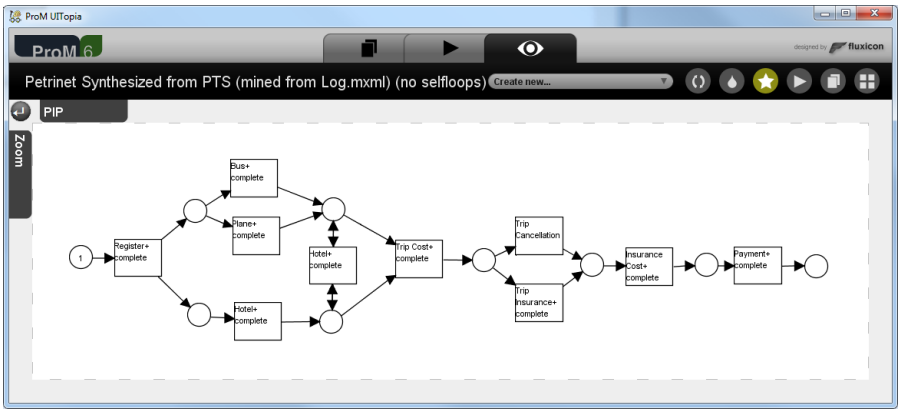


Fig. 37. Result of TS Miner. Note that there are now two transitions referring to hotel bookings (label splitting).

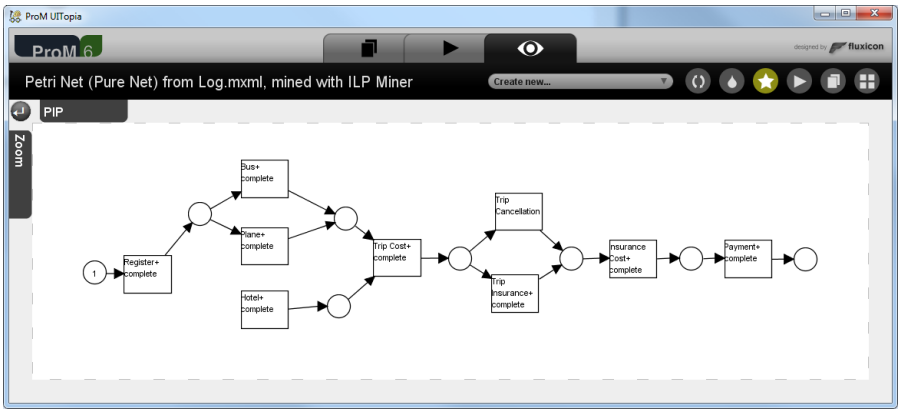


Fig. 38. Result of ILP Miner. The model is able to replay the event log. However, tokens may remain in the place following the hotel booking and bookings can take place before the registration step.

not connect this transition (thus enabling it continuously and destroying the WF-net structure), while the transition system miner introduces two transitions for this step, but it enforces that the second hotel can only be booked after the bus or plane ticket is booked. The ILP miner allows for the hotel booking to occur arbitrarily often, but at least once before the trip costs are calculated.

10 Conclusion

Process mining can be seen as the "missing link" between data mining and traditional model-driven BPM. The spectacular growth of event data is an important enabler for process analysis based on real observations rather than hand-made models only. We have applied ProM in over 100 organizations ranging from municipalities and hospitals to financial institutions and manufacturers of high-tech systems. This illustrates the applicability of the techniques described in this article.

Process mining can be used to diagnose the actual processes. This is valuable because in many organizations most stakeholders lack a correct, objective, and accurate view on important operational processes. However, process mining is not limited to the process discovery techniques mentioned in this article (see for example [2]). Process mining can also be used to improve the discovered processes. Conformance checking can be used for auditing and compliance. By replaying the event log on a process model it is possible to quantify and visualize deviations. Similar techniques can be used to detect bottlenecks and build predictive models. Given the applicability of process mining, we encourage the reader to simply apply the techniques discussed. The event data needed to conduct such experiments can be found in any non-trivial organization. The freely available open-source process mining tool ProM can be downloaded from www.processmining.org and supports all of the process mining techniques mentioned.

In this article we emphasized that four quality dimensions—*fitness*, *simplicity*, *precision*, and *generalization*—need to be balanced [2]. Moreover, we zoomed in on region-based approaches. As shown, conventional *state-based regions* and *language-based regions* focus on fitness and precision, while neglecting simplicity and generalization. Fortunately, it is possible to modify these techniques to also deal with the other two quality dimensions. State-based regions can be used for process discovery tasks provided that the right abstraction is used when constructing the transition system. Language-based regions can be mapped onto an ILP problem where the target function and additional constraints are used to obtain a simple and more general model.

Despite the applicability of process mining there are many interesting challenges; these illustrate that process mining is a young discipline. As discussed, it is far from trivial to construct a process model based on event logs that are incomplete and noisy. Unfortunately, there are still researchers and tool vendors that assume logs to be complete and free of noise. Although heuristic mining, genetic mining, and fuzzy mining provide case-hardened process discovery tech-

niques, many improvements are needed to construct truly intuitive models that are able to explain the most likely/common behavior. Another challenge is to deal with ever-growing datasets, i.e., it is not uncommon to have event logs with millions of cases, billions of events, and thousands of activities [44]. In some cases it is impossible to store all events and process models need to be discovered on-the-fly. In other cases, there is a need to distribute process mining problems over multiple computers. As discussed in [3] this can be done in various ways. Therefore, there are many interesting problems for researchers with a background in Petri nets and eager to analyze processes based on real event data rather than unrealistic toy models.

Acknowledgments. The authors would like to thank all the people that contributed to the development of ProM (www.processmining.org).

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. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer-Verlag, Berlin, 2011.
3. W.M.P. van der Aalst. Distributed Process Discovery and Conformance Checking. In J. de Lara and A. Zisman, editors, *International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*, volume 7212 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, Berlin, 2012.
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*, 2(2):182–192, 2012.
5. W.M.P. van der Aalst, B.F. van Dongen, C.W. Günther, R.S. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H.M.W. Verbeek, and A.J.M.M. Weijters. ProM 4.0: Comprehensive Support for Real Process Analysis. In J. Kleijn and A. Yakovlev, editors, *Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*, volume 4546 of *Lecture Notes in Computer Science*, pages 484–494. Springer-Verlag, Berlin, 2007.
6. W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.
7. W.M.P. van der Aalst, H.A. Reijers, A.J.M.M. Weijters, B.F. van Dongen, A.K. Alves de Medeiros, M. Song, and H.M.W. Verbeek. Business Process Mining: An Industrial Application. *Information Systems*, 32(5):713–732, 2007.
8. 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.
9. 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.
10. 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.

11. W.M.P. van der Aalst and. B. van Dongen, C.W. Günther, A. Rozinat, E. Verbeek, and T. Weijters. ProM: The Process Mining Toolkit. In A.K.A. de Medeiros and B. Weber, editors, *Business Process Management Demonstration Track (BPMDemos 2009)*, volume 489 of *CEUR Workshop Proceedings*, pages 1–4. CEUR-WS.org, 2009.
12. D. Angluin and C.H. Smith. Inductive Inference: Theory and Methods. *Computing Surveys*, 15(3):237–269, 1983.
13. E. Badouel, L. Bernardinello, and P. Darondeau. The Synthesis Problem for Elementary Net Systems is NP-complete. *Theoretical Computer Science*, 186(1-2):107–134, 1997.
14. E. Badouel, L. Bernardinello, and Ph. Darondeau. Polynomial Algorithms for the Synthesis of Bounded Nets. In *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 364–378. Springer-Verlag, Berlin, 1995.
15. E. Badouel and P. Darondeau. Theory of Regions. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 529–586. Springer-Verlag, Berlin, 1998.
16. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Process Mining Based on Regions of Languages. In G. Alonso, P. Dadam, and M. Rosemann, editors, *International Conference on Business Process Management (BPM 2007)*, volume 4714 of *Lecture Notes in Computer Science*, pages 375–383. Springer-Verlag, Berlin, 2007.
17. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Synthesis of Petri Nets from Infinite Partial Languages. In *International Conference on Application of Concurrency to System Design (ACSD 2008)*, pages 170–179. IEEE Computer Society, 2008.
18. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Synthesis of Petri Nets from Scenarios with VipTool. In *Applications and Theory of Petri Nets (Petri Nets 2008)*, volume 5062 of *Lecture Notes in Computer Science*, pages 388–398. Springer-Verlag, Berlin, 2008.
19. R. Bergenthum, J. Desel, S. Mauser, and R. Lorenz. Synthesis of Petri Nets from Term Based Representations of Infinite Partial Languages. *Fundamenta Informaticae*, 95(1):187–217, 2009.
20. 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.
21. 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.
22. J. Carmona, J. Cortadella, and M. Kishinevsky. Genet: A Tool for the Synthesis and Mining of Petri Nets. In *Application of Concurrency to System Design (ACSD 2009)*, pages 181–185. IEEE Computer Society, 2009.
23. J. Carmona, J. Cortadella, and M. Kishinevsky. New Region-Based Algorithms for Deriving Bounded Petri Nets. *IEEE Transactions on Computers*, 59(3):371–384, 2010.
24. J. Carmona, J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. A Symbolic Algorithm for the Synthesis of Bounded Petri Nets. In *Applications and Theory of Petri Nets (Petri Nets 2008)*, pages 92–111, 2008.

25. 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.
26. J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing Petri Nets from State-Based Models. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '95)*, pages 164–171. IEEE Computer Society, 1995.
27. J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri Nets from Finite Transition Systems. *IEEE Transactions on Computers*, 47(8):859–882, August 1998.
28. Ph. Darondeau. Deriving Unbounded Petri Nets from Formal Languages. In *CONCUR 1998*, volume 1466 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.
29. A. Datta. Automating the Discovery of As-Is Business Process Models: Probabilistic and Algorithmic Approaches. *Information Systems Research*, 9(3):275–301, 1998.
30. J. Desel and W. Reisig. The Synthesis Problem of Petri Nets. *Acta Informatica*, 33(4):297–315, 1996.
31. B.F. van Dongen. *Process Mining and Verification*. Phd thesis, Eindhoven University of Technology, 2007.
32. B.F. van Dongen and W.M.P. van der Aalst. Multi-Phase Process Mining: Building Instance Graphs. In P. Atzeni, W. Chu, H. Lu, S. Zhou, and T.W. Ling, editors, *International Conference on Conceptual Modeling (ER 2004)*, volume 3288 of *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, Berlin, 2004.
33. B.F. van Dongen and W.M.P. van der Aalst. Multi-Phase Mining: Aggregating Instances Graphs into EPCs and Petri Nets. In D. Marinescu, editor, *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pages 35–58. Florida International University, Miami, Florida, USA, 2005.
34. B.F. van Dongen, A.K. Alves de Medeiros, and L. Wenn. Process Mining: Overview and Outlook of Petri Net Discovery Algorithms. In K. Jensen and W.M.P. van der Aalst, editors, *Transactions on Petri Nets and Other Models of Concurrency II*, volume 5460 of *Lecture Notes in Computer Science*, pages 225–242. Springer-Verlag, Berlin, 2009.
35. A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures - Part 1 and Part 2. *Acta Informatica*, 27(4):315–368, 1989.
36. R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.
37. E.M. Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, 1967.
38. E.M. Gold. Complexity of Automaton Identification from Given Data. *Information and Control*, 37(3):302–320, 1978.
39. C.W. Günther and W.M.P. van der Aalst. Fuzzy Mining: Adaptive Process Simplification Based on Multi-perspective Metrics. In G. Alonso, P. Dadam, and M. Rosemann, editors, *International Conference on Business Process Management (BPM 2007)*, volume 4714 of *Lecture Notes in Computer Science*, pages 328–343. Springer-Verlag, Berlin, 2007.
40. J. Herbst. A Machine Learning Approach to Workflow Management. In *Proceedings 11th European Conference on Machine Learning*, volume 1810 of *Lecture Notes in Computer Science*, pages 183–194. Springer-Verlag, Berlin, 2000.

41. IEEE Task Force on Process Mining. Process Mining Manifesto. In F. Daniel, K. Barkaoui, and S. Dustdar, editors, *Business Process Management Workshops*, volume 99 of *Lecture Notes in Business Information Processing*, pages 169–194. Springer-Verlag, Berlin, 2012.
42. R. Lorenz. Towards Synthesis of Petri Nets from General Partial Languages. In *German Workshop on Algorithms and Tools for Petri Nets, (AWPN 2008)*, volume 380 of *CEUR Workshop Proceedings*, pages 55–62. CEUR-WS.org, 2008.
43. R. Lorenz and G. Juhás. How to Synthesize Nets from Languages: A Survey. In S.G. Henderson, B. Biller, M. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, editors, *Proceedings of the Wintersimulation Conference (WSC 2007)*, pages 637–647. IEEE Computer Society, 2007.
44. 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.
45. S. Mauser and R. Lorenz. Variants of the Language Based Synthesis Problem for Petri Nets. In *International Conference on Application of Concurrency to System Design (ACSD 2009)*, pages 89–98. IEEE Computer Society, 2009.
46. A.K. Alves de Medeiros, W.M.P. van der Aalst, and A.J.M.M. Weijters. Workflow Mining: Current Status and Future Directions. In R. Meersman, Z. Tari, and D.C. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 389–406. Springer-Verlag, Berlin, 2003.
47. 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.
48. 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.
49. L. Pitt. Inductive Inference, DFAs, and Computational Complexity. In K.P. Jantke, editor, *Proceedings of International Workshop on Analogical and Inductive Inference (AII)*, volume 397 of *Lecture Notes in Computer Science*, pages 18–44. Springer-Verlag, Berlin, 1989.
50. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.
51. 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.
52. A. Schrijver. *Theory of Linear and Integer programming*. Wiley-Interscience, 1986.
53. M. Solé and J. Carmona. Process Mining from a Basis of State Regions. In *Applications and Theory of Petri Nets (Petri Nets 2010)*, volume 6128 of *Lecture Notes in Computer Science*, pages 226–245. Springer-Verlag, Berlin, 2010.
54. M. Solé and J. Carmona. Rbminer: A tool for discovering petri nets from transition systems. In A. Bouajjani and Wei-Ngan Chin, editors, *Automated Technology for Verification and Analysis (ATVA 2010)*, volume 6252 of *Lecture Notes in Computer Science*, pages 396–402. Springer-Verlag, Berlin, 2010.
55. H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing Workflow Processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.
56. H.M.W. Verbeek, J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. ProM 6: The Process Mining Toolkit. In M. La Rosa, editor, *Proc. of BPM Demonstration Track 2010*, volume 615 of *CEUR Workshop Proceedings*, pages 34–39, 2010.

57. H.M.W. Verbeek, J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. XES, XESame, and ProM 6. In P. Soffer and E. Proper, editors, *Information Systems Evolution*, volume 72 of *Lecture Notes in Business Information Processing*, pages 60–75. Springer-Verlag, Berlin, 2010.
58. 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.
59. L. Wen, W.M.P. van der Aalst, J. Wang, and J. Sun. Mining Process Models with Non-Free-Choice Constructs. *Data Mining and Knowledge Discovery*, 15(2):145–180, 2007.
60. 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.