Process Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting

W.M.P. van der Aalst[1], V. Rubin[2,1], H.M.W. Verbeek[1], B.F. van Dongen[1], E. Kindler[3], and C.W. Günther[1]

[1] Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
{w.m.p.v.d.aalst,h.m.w.verbeek,b.f.v.dongen,c.w.gunther}@tue.nl
[2] Software Design and Management (sd&m AG),
Offenbach am Main, Germany.
Vladimir.Rubin@sdm.de
[3] Technical University of Denmark, Informatics and Mathematical Modelling,
Lyngby, Denmark.
eki@imm.dtu.dk

Abstract. Process mining includes the automated discovery of processes from event logs. Based on observed events (e.g., activities being executed or messages being exchanged) a process model is constructed. One of the essential problems in process mining is that *one cannot assume to have seen all possible behavior.* At best, one has seen a representative subset. Therefore, classical synthesis techniques are not suitable as they aim at finding a model that is able to *exactly reproduce the log.* Existing process mining techniques try to avoid such "overfitting" by generalizing the model to allow for more behavior. This generalization is often driven by the representation language and very crude assumptions about completeness. As a result, parts of the model are "overfitting" (allow only what has actually been observed) while other parts may be "underfitting" (allow for much more behavior without strong support for it). None of the existing techniques enables the user to control the balance between "overfitting" and "underfitting". To address this, we propose a two-step approach. First, using a configurable approach, a transition system is constructed. Then, using the "theory of regions", the model is synthesized. The approach has been implemented in the context of ProM and overcomes many of the limitations of traditional approaches.

1 Introduction

More and more information about processes is recorded by information systems in the form of so-called "event logs". A wide variety of *Process-Aware Information Systems* (PAISs) [22] is recording excellent data on actual events taking place. ERP (Enterprise Resource Planning), WFM (WorkFlow Management), CRM (Customer Relationship Management), SCM (Supply Chain Management), and PDM (Product Data Management) systems are examples of such systems. Despite the omnipresence and richness of these event logs, most software vendors use this information for answering only relatively simple questions *under the assumption that the process is fixed and known*, e.g., the calculation of simple performance metrics like utilization and flow time. However, in many domains processes are evolving and people typically have an oversimplified and incorrect view of the actual business processes. Therefore, *process mining* techniques attempt to extract non-trivial and useful information from event logs. One aspect of process mining is *control-flow discovery*, i.e., automatically constructing a process model (e.g., a Petri net) describing the causal dependencies between activities [7, 8, 12, 16, 20, 21, 41]. The basic idea of control-flow discovery is very simple: given an event log containing a set of traces, automatically construct a suitable process model "describing the behavior" seen in the log. Algorithms such as the α-algorithm construct a process model (in this case a Petri net) based on the identification of characteristic patterns in the event log, e.g., one activity always follows another activity.

Research on process mining started by analyzing the logs of WFM systems [7, 8]. These systems have typically excellent logging facilities that allow for a wide variety of process mining techniques. However, discovering the control-flow in such systems is less interesting because the process is controlled based on an already known process model. Moreover, WFM systems are just one type of systems in a broad spectrum of systems recording events. To illustrate this, we provide some examples of processes in non-workflow environments that are being recorded today:

- For many years, hospitals have been working towards a comprehensive Electronic Patient Record (EPR), i.e., information about the health history of a patient, including all past and present health conditions, illnesses, and treatments. Although there are still many problems that need to be resolved (mainly of a non-technical nature), many people forget that most of this information is already present in today's hospital information systems. For example, by Dutch law all hospitals need to record the diagnosis and treatment steps at the level of individual patients in order to receive payment. This so-called "Diagnose Behandeling Combinatie" (DBC) forces hospitals to record all kinds of events.
- Today, many organizations are moving towards a Service-Oriented Architecture (SOA). A SOA is essentially a collection of services that communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. Here, technologies and standards such such as SOAP, WSDL, and BPEL are used. It is relatively easy to listen in on the message exchange between services. This results in massive amounts of relevant information that can be recorded.
- Increasingly, professional high-tech systems such as high-end copiers, complex medical equipment, lithography systems, automated production systems, etc. record events which allow for the monitoring of these systems. These raw event logs can be distributed via the internet allowing for both

real-time and off-line analysis. This information is valuable for (preventive) maintenance, monitoring user adoption, etc.
- Other examples can be found in the classical administrative systems of large organizations using e.g. ERP, CRM, SCM, and PDM software. Consider for example processes in banks, insurance companies, local governments, etc. Here most activities are recorded in some form.

These examples illustrate that one can find a variety of event logs in today's PAISs. However, in most cases real processes are not as simple and structured as the processes typically supported by WFM systems. Most process mining algorithms produce spaghetti-like diagrams that do not correspond to valid process models (e.g., the models have deadlocks, etc.) and that do not provide a lot of insight. For example, the well-known α-algorithm [7] connects two activities a and b if a is sometimes followed by b but never the other way around. This may lead to many connections that do not reflect the actual causal dependencies. In fact, existing process mining algorithms tend to:

- *Have problems with complex control-flow constructs.* For example, many process mining algorithms are unable to deal with non-free-choice constructs and complex nested loops.
- *Not allow for duplicates (i.e., occurrences of the same activity in different phases of the process).* In the event log it is not possible to distinguish between activities that are logged in a similar way, i.e., there are multiple activities that have the same "footprint" in the log. As a result, most algorithms map these different activities onto a simple activity thus making the model incorrect or counter-intuitive.
- *Yield inconsistent models.* For more complicated processes, many algorithms have a tendency to produce models that may have deadlocks and/or livelocks. It seems vital that the generated models satisfy some soundness requirements (e.g., the soundness property defined in [1]). Unfortunately, most algorithms may produce models that are syntactically or semantically incorrect, e.g., models with a deadlock or not allowing for behavior seen in the log.
- *Have problems balancing between "overfitting" and "underfitting".* Some algorithms have a tendency to over-generalize, i.e., the discovered model allows for much more behavior than actually recorded in the log. The reason for over-generalizing is often the representation used and a coarse completeness notion. Other algorithms have a tendency to "overfit" the model. Classical synthesis approaches such as the "theory of regions" aim at a model that is able to exactly reproduce the log. Therefore, the model is merely another representation of the log without deriving any new knowledge.

This paper will *address all four problems but primarily focus on the last problem.* We aim at creating a balance between "overfitting" and "underfitting", therefore, we first define these two notions. Let L be a log and M be a model.

- *M is overfitting L* if M does not generalize and is sensitive to particularities in L. In an extreme case, M could merely be a representation of the log without any inference. A mining algorithm is producing overfitting models if the removal or addition of a small percentage of the process instances in L would lead to a remarkably different model. In a complex process with many possible paths, most process instances will follow a path not taken by other instances in the same period. Therefore, it is undesirable to construct a model that allows only for the paths that happened to be present in the log as this is only a fraction of all possible paths. The best way to avoid overfitting is to use many process instances. However, if one knows that only a fraction of the possible event sequences are in the log, the only way to avoid overfitting is to generalize and have a model M that allows for more behavior than recorded in L.

– *M is underfitting L* if M allows for "too much behavior" that is not supported by L. It is very easy to construct a model that allows for the behavior seen in the log but also completely different behavior. For example, assume a log L consisting of 1000 cases. For each case A is followed by B and there are no cases where B is followed by A. Obviously, one could derive a causal dependency between A and B. However, one could also create a model M where A and B are in parallel. The latter would be not be "wrong" in the sense that the behavior seen in the log is possible according to the model. However, it is very unlikely and therefore we one could argue that M is underfitting L.

To illustrate the problem between overfitting and underfitting, consider some process in a hospital. When observing such a process over a period of years it is very likely that many patients follow a "unique process", i.e., seen from the viewpoint of a particular patient it is very unlikely that there is another patient that has exactly the same sequence of events. Therefore, it does not make sense to assume that the event log contains all possible paths a particular case can take. In fact, it is very likely that the next patient will have a sequence of events different from all earlier patients. Therefore, one cannot assume that an event log is "complete" and one is forced to generalize to avoid overfitting. However, at the same time underfitting ("anything is possible") should be avoided.

This paper will present a *new type of process discovery* which uses a *two-step* approach: (1) we generate a transition system that is used as an intermediate representation and (2) based on this we obtain a Petri net contructed through regions [9,10,14,15,23] as a final representation. Transition systems are the most basic representation of processes, but even simple processes tend to have many states (cf. "state explosion" problem in verification). However, using the "theory of regions" and tools like Petrify [15], transition systems can be "folded" into more compact representations, e.g., Petri nets [17,34]. Especially transition systems with a lot of concurrency (assuming interleaving semantics) can be reduced dramatically through the folding of states into regions, e.g., transition systems with hundreds or even thousands of states can be mapped onto compact Petri nets. However, before using regions to fold transition systems into Petri nets, we first need to derive a transition system from an event log. This paper shows that this can be done in several ways *enabling a repertoire of process discovery* *approaches.* Different strategies for generating transition systems are possible depending on the desired degree of generalization, i.e., we will show that while constructing the transition system it is possible to control the degree and nature of generalization and thus *allow the analyst to balance between "overfitting" and "underfitting"*.

The two-step approach presented in this paper has been implemented in ProM (`www.processmining.org`). ProM serves as a testbed for our process mining research [2]. For the second step of our approach ProM calls Petrify [15] to synthesize the Petri net.

The remainder of this paper is organized as follows. Section 2 provides an overview of process mining and discusses problems related to process discovery. The first step of our approach is presented in Section 3. Here it is shown that there are various ways to construct a transition system based on a log. This results in a family of process mining techniques that assist in finding the balance between "overfitting" and "underfitting". The second step where the transition system is transformed into a Petri net is presented in Section 4. Section 5 describes the implementation, evaluation, and application of our two-step approach. Related work is discussed in Section 6, and Section 7 concludes the paper.

2 Process Mining

This section introduces the concept of process mining and provides examples of issues related to control-flow discovery. It also discusses requirements such as the need to produce correct models and to balance between models that are too specific or too generic.

2.1 Overview of Process Mining

As indicated in the introduction, today's information systems are recording events in so-called event logs. The goal of process mining is to extract information on the process from these logs, i.e., process mining describes a family of *a-posteriori* analysis techniques exploiting the information recorded in the event logs. 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).

Process mining addresses the problem that most organizations have very limited information about what is actually happening in their organization. In practice, there is often a significant gap between what is prescribed or supposed to happen, and what *actually* happens. Only a concise assessment of the organizational reality, which process mining strives to deliver, can help in verifying process models, and ultimately be used in a process redesign effort.

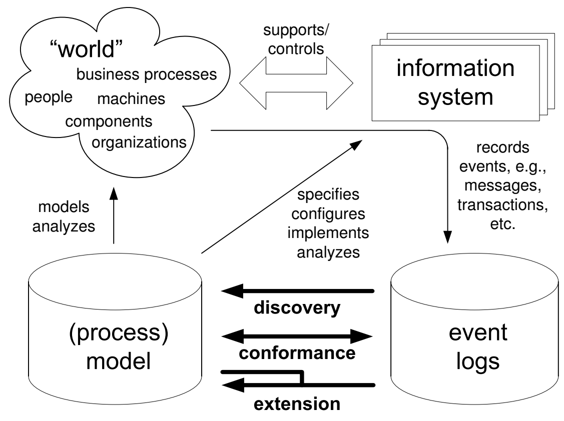

Fig. 1. Three types of process mining: (1) Discovery, (2) Conformance, and (3) Extension.

The idea of process mining is to discover, monitor and improve real processes (i.e., not assumed processes) by extracting knowledge from event logs. We consider three basic types of process mining (Figure 1):

- **Discovery**: There is no a-priori model, i.e., based on an event log some model is constructed. For example, using the α-algorithm [7] a process model can be discovered based on low-level events.
- **Conformance**: There is an a-priori model. This model is used to check if reality, as recorded in the log, conforms to the model and vice versa. For example, there may be a process model indicating that purchase orders of more than one million Euro require two checks. Another example is the checking of the four-eyes principle. Conformance checking may be used to detect deviations, to locate and explain these deviations, and to measure the severity of these deviations. An example, is the conformance checking algorithms described in [37].
- **Extension**: There is an a-priori model. This model is extended with a new aspect or perspective, i.e., the goal is not to check conformance but to enrich the model. An example is the extension of a process model with performance data, i.e., some a-priori process model is used on which bottlenecks are projected. Another example is the decision mining algorithm described in [36] that extends a given process model with conditions for each decision.

Today, process mining tools are becoming available and are being integrated into larger systems. The ProM framework [2] provides an extensive set of analysis techniques which can be applied to real process enactments while covering the whole spectrum depicted in Figure 1. ARIS PPM was one of the first commercial tools to offer some support for process mining. Using ARIS PPM, one can extract performance information and social networks. Also some primitive form of process discovery is supported. However, ARIS PPM still requires some a-priori modeling. The BPM|suite of Pallas Athena was the first commercial tool to support process discovery without a-priori modeling. Although the above tools can already be applied to real-life processes, it remains a challenge to extract suitable process models from event logs.

2.2 Control-Flow Discovery

The focus of this paper is on *control-flow discovery*, i.e., extracting a process model from an event log. The event logs of various systems may look very different. Some systems log a lot of information while other systems provide only very basic information. In fact, in many cases one needs to extract event logs from different sources and merge them. Tools such as our ProM Import Framework allows developers to quickly implement plug-ins that can be used to extract information from a variety of systems and convert this into the so-called MXML format [25]. MXML encompasses timestamps (when the event took place), originators (which person or software component executed the corresponding activity), transactional data, case data, etc. 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. The only requirement that we assume in this paper is that *any event needs to be linked to a case (process instance) and an activity.* Assuming that only this information is available, an event is described by a pair (c, a) where c refers to the case and a refers to the activity. In process mining, one typically abstracts from dependencies between cases. Hence, we assume that each case is executed independently from other cases, i.e., the routing of one case does not depend on the routing of other cases (although they may compete for the same resources). As a result, *we can focus on the ordering of activities within individual cases.* Therefore, a single case σ can be represented as a sequence of activities, i.e., a trace $\sigma \in A^*$ where A is the set of activities. Consequently, a log can be seen as a collection of traces (i.e., $L \subseteq A^*$).

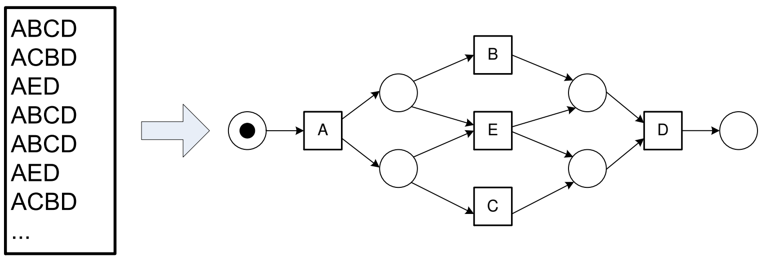

Fig. 2. A log represented by sequences of activities and the process model that is discovered using the α-algorithm.

Figure 2 shows an example of a log and the corresponding process model that can be discovered using techniques such as the α-algorithm [7]. It is easy to see that the Petri net is able to reproduce the log, i.e., there is a good fit between the log and the discovered process model.[4] Note that the α-algorithm is a very simple algorithm. Unfortunately, like many other algorithms, it has several limitations. In Section 6, we mention some of these algorithms.

As indicated in the introduction, existing process mining algorithms for control-flow discovery typically have several problems. Using the example shown in Figure 2, we can discuss these problems in a bit more detail.

The first problem is that many algorithms have problems with *complex control-flow constructs*. For example, the choice between the concurrent execution of B and C or the execution of just E shown in Figure 2 cannot be handled by many algorithms. Most algorithms do *not* allow for so-called "non-free-choice constructs" where concurrency and choice meet. The concept of free-choice nets is well-defined in the Petri net domain [17]. However, in reality processes tend to be non-free-choice. In the example of Figure 2, the α-algorithm [7] is able to deal with the non-free-choice construct. However, it is easy to think of a non-free-choice process that cannot be discovered by the α-algorithm. The non-free-choice construct is just one of many constructs that existing process mining algorithms have problems with. Other examples are arbitrary nested loops, unbalanced splits and joins, partial synchronization, etc. In this context it is important to note that *process mining is, by definition, restricted by the expressive power of the target language*, i.e., if a simple or highly informal language is used, process mining is destined to produce less relevant or over-simplified results.

The second problem is the fact that most algorithms have problems with *duplicates*. The same activity may appear at different places in the process or different activities may be recorded in an indistinguishable manner. Consider for example Figure 2 and assume that activities A and D are both recorded as X (or, equivalently, assume that A and D are both replaced by activity X). Hence the trace $ABCD$ in the original model is recorded as $XBCX$. Most algorithms will try to map the first and the second X onto the same activity. In some cases this make sense, e.g., to create loops. However, if the two occurrences of X (i.e., A and D) really play a different role in the process, then algorithms that are unable to separate them will run into all kinds of problems, e.g., the model becomes more difficult or incorrect. Since the duplicate activities have the same "footprint" in the log, most algorithms map these different activities onto a single activity thus making the model incorrect or counter-intuitive.

The third problem is that many algorithms have a tendency to generate *inconsistent models*. Note that here we do not refer to the relation between the log and the model but to the internal consistency of the model by itself. For example, the α-algorithm [7] may yield models that have deadlocks or livelocks when the log shows certain types of behavior. When using Petri nets as a model to represent processes, an obvious choice is to require the model to be *sound* [1].

[4] In this paper, we assume that the reader has a basic understanding of Petri nets, cf. [17, 19, 34].

Soundness implies that for any case: (1) the model can potentially terminate from any reachable state (option to complete), (2) that the model has no dead parts, and (3) that no tokens are left behind (proper completion). See [1, 7] for details.

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

The four problems just mentioned illustrate the need for more powerful algorithms. See also [32] for a more elaborate discussion on these and other challenges in control-flow discovery.

2.3 Notions of Completeness

When it comes to process mining the notion of *completeness* is very important. Like in any data mining or machine learning context one cannot assume to have seen all possibilities in the "training material" (i.e., the event log at hand). In Figure 2, the set of possible traces found in the log is exactly the same as the set of possible traces in the model, i.e., $\{ABCD, ACBD, AED\}$. In general, this is not the case. For example, the trace $ABECD$ may be possible but did not (yet) occur in the log.

To define the concept of *completeness* assume that there is a model correctly describing the process being observed. Let L be the set of traces in some event log and L_M the set of all traces possible according to the model. Clearly, $L \subseteq L_M$. If $L = L_M$, the log is trivially complete. However, as indicated above one can never assume $L = L_M$ because, typically, $|L|$ is much smaller than $|L_M|$. For a model with lots of choices and concurrency $|L|$ is a fraction of $|L_M|$. Therefore, it makes no sense to define completeness as $|L|/|L_M|$. Therefore, other criteria are needed to describe how "complete" a log is. For example, the α-algorithm [7] assumes that the log is "locally complete", i.e., if there are two activities X and Y, and X can be directly followed by Y this should be observed in the log. Other completeness notions are possible and based on these notions one can reason about the correctness of a mining algorithm [7].

To illustrate the relevance of completeness, consider 10 tasks which can be executed in parallel. The total number of interleavings is $10! = 3628800$ (i.e., $|L_M| = 3628800$). It is probably not realistic that each interleaving is present in the log, since typically $|L| << |L_M|$. Moreover, even if $|L|$ and $|L_M|$ are of the same order of magnitude, it is still very unlikely that $L = L_M$. To motivate this consider the following analogy. In a group of 365 people it is very unlikely that everyone has a different birthdate ($365!/365^{365}$). Similarly, it is unlikely that all possible traces will occur for process of some complexity. However, for local completeness as assumed by the α-algorithm [7] only $10(10-1) = 90$ different observations are needed.

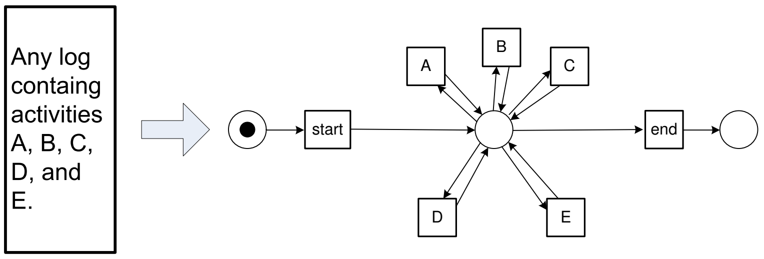

Fig. 3. The so-called "flower Petri net" allowing for any log containing *A*, *B*, *C*, *D*, and *E*.

Completeness is closely linked to the notions of *overfitting* and *underfitting* mentioned earlier. It is also linked to Occam's Razor, a principle attributed to the 14th-century English logician William of Ockham. The principle states that "one should not increase, beyond what is necessary, the number of entities required to explain anything", to look for the "simplest model" that can explain what is in the log. Using this principle different algorithms assume different notions of completeness.

Process mining algorithms needs to strike a balance between "overfitting" and "underfitting". A model is overfitting if it does not generalize and only allows for the exact behavior recorded in the log. This means that the corresponding mining technique assumes a very strong notion of completeness: "If the sequence is not in the event log, it is not possible.". An underfitting model over-generalizes the things seen in the log, i.e., it allows for more behavior even when there are no indications in the log that suggest this additional behavior. An example is shown in Figure 3. This so-called "flower Petri net" allows for any sequence starting with *start* and ending with *end* and containing any ordering of activities A, B, C, D, and E in between. Clearly, this model allows for the set of traces $\{ABCD, ACBD, AED\}$ (without the added *start* and *end* activities) but also many more, e.g., $DDAA$, without much evidence that they should be possible.

Let us now consider another example showing that it is difficult to balance between being too general and too specific. Figure 4 shows two event logs and two models. Both logs are possible according to the model shown in (d). However, only log (a) is possible according to the model shown in (c) because this model does not allow for ACE and BCD present in log (b). Clearly, (c) seems to be a suitable model for (a), and (d) seems to be a suitable model for (b). However, the question is whether (d) is also a suitable model for (a). If there are just two cases ACD and BCE, then there is no reason to argue why (d) would not be a suitable

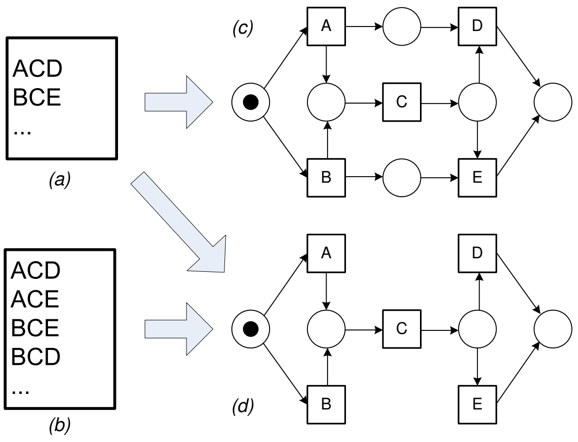

Fig. 4. Two logs and two models illustrating issues related to completeness (i.e., "overfitting" and "underfitting").

model. However, if there are 100 cases following ACD and 100 cases BCE, then it is difficult to justify (d) as a suitable model and (c) seems more appropriate. If ACE and BCD are indeed possible (model (d)), then it seems unlikely that they never occurred in one of the 200 cases. This shows that sometimes it is not only relevant what is in the log, but also how often it is in the log.

Figure 4 shows that there is a delicate balance and that it is non-trivial to compare logs and process models. In [35] notions such as *fitness* and *appropriateness* have been quantified. An event log and Petri net "fit" if the Petri net can generate each trace in the log.[5] In other words: the Petri net should be able to "parse" (i.e., reproduce) every activity sequence observed. In [35] it is shown that it is possible to quantify fitness as a measure between 0 and 1. The intuitive meaning is that a fitness close to 1 means that all observed events can be explained by the model. However, the precise meaning is more involved since tokens can remain in the net and not all transactions in the model need to be logged [35]. Unfortunately, a good fitness alone does not imply that the model is indeed suitable, e.g., it is easy to construct Petri nets that are able to reproduce any event log (cf. the "flower model" in Figure 3). Although such Petri nets have a fitness of 1, they do not provide meaningful information. Therefore, in [35] a second dimension is introduced: *appropriateness*. Appropriateness tries to answer the following question: "Does the model describe the observed process in a concise way?". This notion can be evaluated from both a *structural* and a *behavioral* perspective. In [35] it is shown that a "good" process model should somehow be minimal in structure to clearly reflect the described behavior, referred to as *structural appropriateness*, and minimal in behavior in order

[5] It is important not to confuse *fitness* with *overfitting* and *underfitting*. A model that is overfitting or underfitting may have a fitness of 1.

to represent as closely as possible what actually takes place, which will be called *behavioral appropriateness*. The ProM conformance checker supports both the notion of fitness and various notions of appropriateness, i.e., for a given log and a given model it computes the different metrics.

Although there are different ways to quantify notions such as fitness and appropriateness, it is difficult to agree on the definition of an "optimal model". What is optimal seems to depend on the intended purpose and even given a clear metric there may be many models having the same score. *Since there is not "one size fits all", it is important to have algorithms that can be tuned to specific applications.* Therefore, we present an approach that allows for different strategies enabling different interpretations of completeness to avoid overfitting and underfitting.

Linked to notions such as completeness, overfitting, and underfitting is the issue of *noise*. The log may contain traces that one would like to refer to as noise, e.g., incorrectly logged events (i.e., the log does not reflect reality) and exceptions (i.e., sequences of events corresponding to "abnormal behavior"). The fact that a particular trace of events is observed does not automatically mean that the model should be able to reproduce it. Noise is typically tackled by cleaning the log and setting thresholds [3, 31, 41]. This paper will not address issues related to noise. However, existing ideas for dealing with noise [3, 31, 41] can easily be combined with the approach presented here.

3 Constructing a Transition System (Step 1)

After introducing the concept of control-flow discovery and discussing the problems of existing approaches, we can now explain our two-step approach. In the first step, we construct a transition system and in the second step a Petri net is synthesized from this transition system. This section describes the first and most important step. An important quality of the first step is that, unlike existing approaches, it can be tuned towards the application. Depending on the desired properties of the model and the characteristics of the log, the algorithm can be tuned to provide a more suitable model.

3.1 Preliminaries

To explain the different strategies for constructing transition systems from event logs, we need the following notations.

$f \in A \rightarrow B$ is a function with domain A and range B. $f \in A \not\rightarrow B$ is a partial function, i.e., the domain of f may be a subset of A.

Let A be a set. $\mathbb{B}(A) = A \rightarrow \mathbb{N}$ is the set of multi-sets (bags) over a finite domain A, i.e., $X \in \mathbb{B}(A)$ is a multi-set, where for each $a \in A$: $X(a)$ denotes the number of times a is included in the multi-set. The sum of two multi-sets $(X+Y)$, the difference $(X - Y)$, the presence of an element in a multi-set $(x \in X)$, and the notion of subset $(X \leq Y)$ are defined in a straightforward way. Moreover, we also apply these operators to sets, where we assume that a set is a multiset in which every element occurs exactly once. The operators are also robust with respect to the domains of the multi-sets, i.e., even if X and Y are defined on different domains, $X + Y$, $X - Y$, and $X \leq Y$ are defined properly by extending the domain where needed. $|X| = \sum_{a \in A} X(a)$ is the cardinality of some multi-set X over A. $set(X)$ transforms a bag X into a set: $set(X) = \{a \in X \mid X(a) > 0\}$. We will use the following notation to denote a concrete multiset: $\{a, b^2, c^3, d\}$ is the multiset with seven elements: one a, two b's, three c's, and one d.

$\mathcal{P}(A)$ is the powerset of A, i.e., $\mathcal{P}(A) = \{X \mid X \subseteq A\}$.

For a given set A, A^* is the set of all finite sequences over A. A finite sequence over A of length n is a mapping $\sigma \in \{1, \ldots, n\} \rightarrow A$. Such a sequence is represented by a string, i.e., $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ where $a_i = \sigma(i)$ for $1 \leq i \leq n$. $hd^k(\sigma) = \langle a_1, a_2, \ldots, a_{k\ min\ n} \rangle$, i.e., the sequence consisting of the first k elements (if possible). Note that $hd^0(\sigma)$ is the empty sequence and for $k \geq n$: $hd^k(\sigma) = \sigma$. $tl^k(\sigma) = \langle a_{(n-k+1)\ max\ 1}, a_{k+2}, \ldots, a_n \rangle$, i.e., sequence composed of the last k elements (if possible). Note that $tl^0(\sigma)$ is the empty sequence and for $k \geq n$: $tl^k(\sigma) = \sigma$. $\sigma \uparrow X$ is the projection of σ onto some subset $X \subseteq A$, e.g., $\langle a, b, c, a, b, c, d \rangle \uparrow \{a, b\} = \langle a, b, a, b \rangle$ and $\langle d, a, a, a, a, a, d \rangle \uparrow \{d\} = \langle d, d \rangle$.

For any sequence σ over A, the Parikh vector $par(\sigma)$ maps every element a of A onto the number of occurrences of a in σ, i.e., $par(\sigma) \in \mathbb{B}(A)$ where for any $a \in A$: $par(\sigma)(a) = |\sigma \uparrow \{a\}|$.

Later, we will use the Parikh vector to count the number of times an activity occurs in a log trace.

3.2 Basic Approach

Although an event log can store transactional information, information about resources, related data, timestamps, etc. we first focus on the ordering of activities. Cases are executed independently from each other, and therefore, we can simply restrict our input to the ordering of activities within individual cases. A single case is described as a sequence of activities and a log can be described as a set of traces[6].

Definition 1 (Simple Trace, Simple Event log). *Let A be a set of activities. $\sigma \in A^*$ is a (simple)* trace *and $L \in \mathcal{P}(A^*)$ is a (simple)* event log*.*

The reason that we call $\sigma \in A^*$ a *simple* trace and $L \in \mathcal{P}(A^*)$ a *simple* event log is that we initially assume that an event only refers to the activity being executed. In Section 3.4 we will refine this view and include attributes describing other perspectives (e.g., data, time, resources, etc.).

The set of activities can be found by inspecting the log. However, 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,

[6] Note that we ignore multiple occurrences of the same trace in this paper. When dealing with issues such as noise, it is vital to also look at the frequency of activities and traces. Therefore, an event log is typically defined as a multi-set of traces rather than a set. However, for the purpose of this paper it suffices to consider sets.

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 paper, we propose to *define states explicitly* and start with the definition of a transition system.

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 define 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.

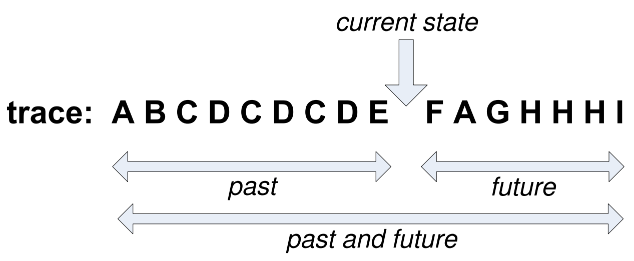

Fig. 5. Three basic "ingredients" can be considered as a basis for calculating the "process state": (1) past, (2) future, and (3) past and future.

Figure 5 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 6. Here we start from the same log as used in Figure 2. If we just consider the prefix (i.e., the past), we get the transition system shown in Figure 6(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 that lead to different states, e.g., executing activity B results in state $\langle A, B\rangle$, etc. Note that in Figure 6(a) there is one initial state and three final states. Figure 6(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 6(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.

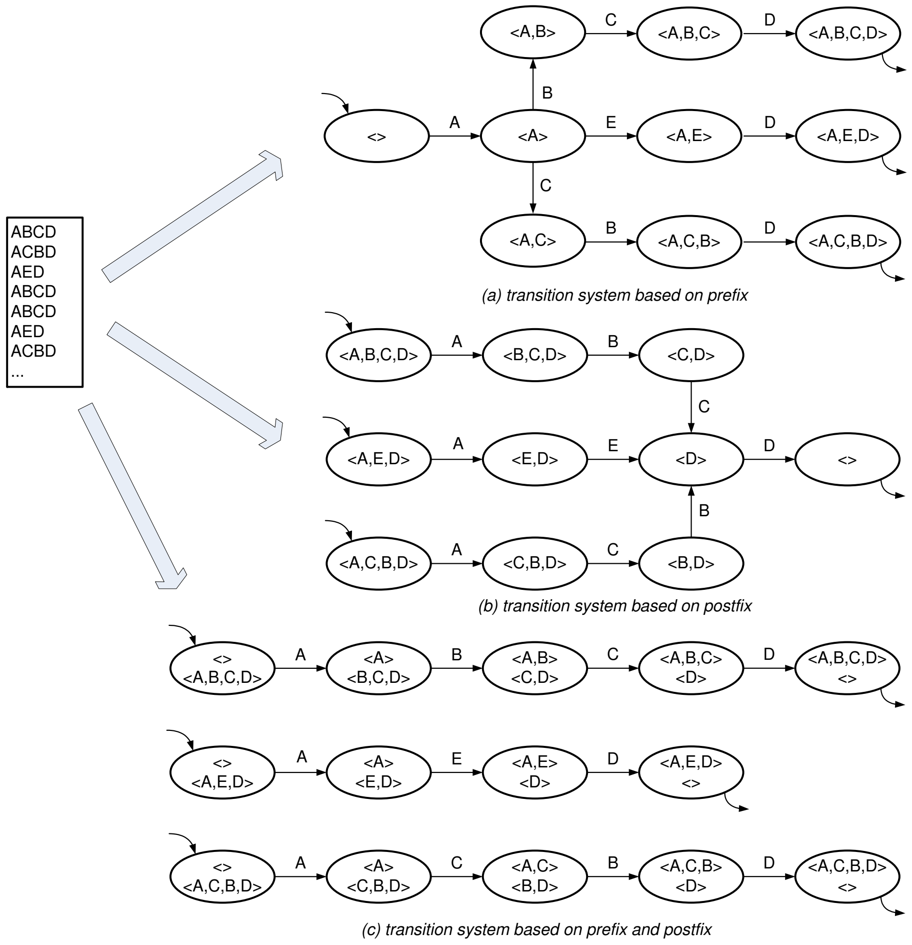

Fig. 6. Three transition systems derived from the log.

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). However, many *abstractions* are possible as shown below. The abstractions can be applied to prefixes, postfixes, or both.

Abstraction 1: Maximal horizon (h) The basis of the state calculation can be the complete prefix (postfix) or a partial prefix (postfix). In the later case, only a subset of the trace is considered. For example, instead of taking the complete prefix $\langle A, B, C, D, C, D, C, D, E \rangle$ shown in Figure 5, only the last four ($h = 4$) events could considered: $\langle D, C, D, E \rangle$. In a partial prefix, only the h most recent events are considered as input for the state calculation. In a partial postfix, also a limited horizon is considered, i.e., seen from the state under consideration, only the next h events are taken into account. Taking a complete prefix (postfix) corresponds to $h = \infty$.

Abstraction 2: 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. Filtering can be seen as projecting the horizon onto a set of activities F. For example, if $F = \{C, D\}$, then the prefix $\langle A, B, C, D, C, D, C, D, E \rangle$ shown in Figure 5 is reduced to $\langle C, D, C, D, C, D \rangle$. Note that the filtering is applied to the sequence resulting from the horizon. It is also possible to first filter the log, but we consider this to be part of the preprocessing of the log and not part of the mining algorithm itself. The occurrence of some activity $a \in F$ is considered relevant for the state of a case. If $a \notin F$, then the occurrence of a is still relevant for the process (i.e., it may appear on the arcs in the transition system) but is assumed to be irrelevant for determining the state. If a is not relevant at all, it should be filtered out before and should not appear in L.

Abstraction 3: 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. The number m determines the maximum number of filtered events. Consider the prefix $\langle A, B, C, D, C, D, C$ $, D, E \rangle$ shown in Figure 5. Suppose that $h = 6$, then the first abstraction yields $\langle D, C, D, C, D, E \rangle$. Suppose that $F = \{C, E\}$, then the second abstraction yields $\langle C, C, E \rangle$. Suppose that $m = 2$, then the third abstraction yields $\langle C, E \rangle$. Note that there is a difference between h and m. If $h = 2$, $F = \{C, E\}$, and $m = 6$, then the result is $\langle E \rangle$ rather than $\langle C, E \rangle$. Note that $m = \infty$ implies that no events are removed by this third abstraction.

Abstraction 4: 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. For the current state it may be less interesting to know when some activity a occurred and how many times a occurred, i.e., only the fact that it occurs within the scope determined by the first three abstractions is relevant. In other cases, it may be relevant to know how many times a occurred or it may be essential to know whether a occurred before b or not. This suggests that there are three ways of representing knowledge about the past and the future:

- *sequence*, i.e., the order of activities is recorded in the state,
- *multi-set of activities*, i.e., the number of times each activity is executed ignoring their order, and
- *set of activities*, i.e., the mere presence of activities.

Consider again the prefix $\langle A, B, C, D, C, D, C, D, E\rangle$ and suppose that $h = \infty$, $F = A$, and $m = \infty$, then the fourth abstraction step yields $\langle A, B, C, D, C, D, C, D, E\rangle$ (sequence), $\{A, B, C^3, D^3, E\}$ (multiset), and $\{A, B, D, E\}$ (set). We will denote this abstraction using the identifier q, i.e., $q = seq$ (sequence), $q = ms$ (multiset), or $q = set$ (set).

Abstraction 5: 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. Note that the arcs are not removed from the transition system; only the label on the arc is suppressed. This abstraction is particularly useful if there are many activities having a similar effect in terms of changing states. Rather than having many arcs from one state to another, these are then collapsed into a single unlabeled arc.

Figure 7 illustrates the abstractions. In Figure 7(a) only the set abstraction is used $q = set$. The result is that several states are merged (compare with Figure 6(a)). In Figure 7(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 7(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.

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 while the last abstraction is applied to the resulting transition system. As a result of these choices, many different transitions systems can be generated. If more 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 6(c) where each trace is presented separately without deducing control-flow constructs. In fact, all of the abstractions used in Figure 6 will lead to overfitting because the whole prefix and/or postfix is considered.

At first it may seem confusing that there are multiple process models that can be deduced based on the same log, however, as indicated in the introduction it

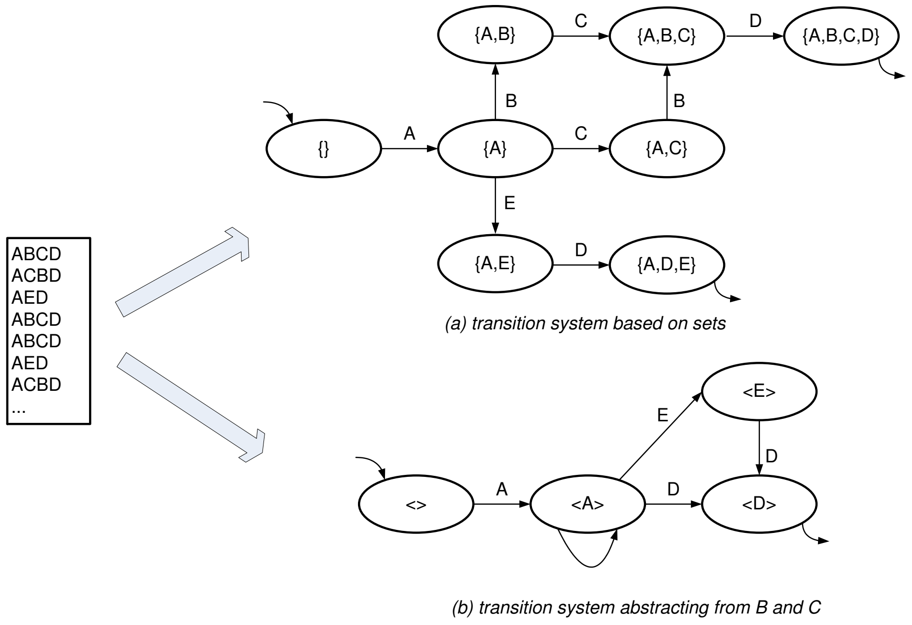

Fig. 7. Two transition systems using the following prefix abstractions: (a) $h = \infty$, $F = A$ (i.e., all activities), $m = \infty$, $q = set$, and $V = A$, and (b) $h = \infty$, $F = \{A, D, E\}$, $m = 1$, $q = seq$, and $V = \{A, D, E\}$.

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" in a controlled way. Existing approaches do not allow the analyst to control the degree and nature of abstraction, i.e., the degree of generalization is fixed by the method.

3.3 Formalization Basic Approach

Let us now further formalize the ideas presented so far. For this purpose, we first take a broader perspective and then focus on the concrete abstractions discussed thus far.

To determine the states of the transition system, we need to construct a so-called state representation based on the first four abstractions and the choice of prefix and postfix.

Definition 2 (State representation). *A state representation function state() is a function that, given a sequence σ and a number k indicating the number of events of σ that have occurred, produces some representation r. Formally, $state \in (A^* \times I\!N) \not\rightarrow R$ where A is the set of activities, R is the set of possible* *state representations (e.g., sequences, sets, or bags over A), and $dom(state) = \{(\sigma, k) \in A^* \times I\!N \mid 0 \leq k \leq |\sigma|\}$.*

Based on the notion of a $state()$ function, we can define the transition system. In this definition we use a renaming function j_V that renames invisible activities are renamed to τ: $j_V(a) = a$ if $a \in V$ and $j_V(a) = \tau$ otherwise. Such transitions are not labeled in the diagram, e.g., see Figure 7(b) where the B and C labels are not shown.

Definition 3 (Transition system). *Let A be a set of activities and let $L \in \mathcal{P}(A^*)$ be an* event log. *Given a $state()$ function as defined before and a set of visible activities $V \subseteq A$, we define a* labeled transition system *$TS = (S, E, T)$ where $S = \{state(\sigma, k) \mid \sigma \in L \ \wedge \ 0 \leq k \leq |\sigma|\}$ is the state space, $E = V \cup \{\tau\}$ is the set of events (labels) and $T \subseteq S \times E \times S$ with $T = \{(state(\sigma, k), j_V(\sigma(k+1)), state(\sigma, k+1)) \mid \sigma \in L \ \wedge \ 0 \leq k < |\sigma|\}$ is the transition relation. $S^{start} \subseteq S$ is the set of initial states, i.e., $S^{start} = \{state(\sigma, 0) \mid \sigma \in L\}$. $S^{end} \subseteq S$ is the set of final states, i.e., $S^{end} = \{state(\sigma, |\sigma|) \mid \sigma \in L\}$.*

The set of states of the transition system is determined by the range of function $state()$ when applied to the log data. The transitions in the transition system have a label in $E = V \cup \{\tau\}$. Note that V is the set of visible activities and τ refers to activities made "invisible" in the transition system.

The *algorithm* for constructing a transition system is straightforward: for every trace σ, iterating over k $(0 \leq k \leq |\sigma|)$, we create a new state $state(\sigma, k)$ if it does not exist yet. Then the traces are scanned for transitions $state(\sigma, k-1) \xrightarrow{j_V(\sigma(k))} state(\sigma, k)$ and these are added if they do not exist yet[7]. Recall that, if $j_V(\sigma(k)) = \tau$, then the label is not shown in the diagram.

So given a $state()$ function and a set of visible activities V it is possible to automatically build a transition system. This was already illustrated in Figure 7 which shows two examples using the same log but different choices for $state()$ and V.

Let us now consider the construction of different $state()$ functions. To this end, this we introduce some notations. First, we show how to obtain the past and future of a case σ after k steps.

Definition 4 (Past and future of a case). *Let A be a set of activities and let $\sigma = \langle a_1, a_2, \ldots, a_n \rangle \in A^*$ be a trace that represents a complete execution of a case. The* past *of this case after executing k steps $(0 \leq k \leq n)$ is $hd^k(\sigma)$. The* future *of this case after executing k steps $(0 \leq k \leq n)$ is $tl^{n-k}(\sigma)$. The past and future are denoted as a pair: $(hd^k(\sigma), tl^{n-k}(\sigma))$.*

Note that $\sigma = hd^k(\sigma) tl^{n-k}(\sigma)$, i.e., the concatenation of past and future yields the whole trace.

Let us now consider the first four abstractions presented in Section 3.2. For simplicity, we first focus on the past of a case. Let $\sigma_0 = hd^k(\sigma)$ be the complete prefix of some case σ after k steps.

[7] Note that the elements of T are often denoted as $s_1 \xrightarrow{e} s_2$ instead of (s_1, e, s_2).

The first abstraction presented in Section 3.2 can be tackled using function *tl*. Recall that this abstraction sets a horizon of length h. Assuming a horizon h, the result of this first abstraction is $\sigma_1 = tl^h(\sigma_0)$. The second abstraction can be tackled using the projection operator $\uparrow$ defined earlier. Assuming a filter F, the result of this second abstraction is $\sigma_2 = \sigma_1 \uparrow F$. The third abstraction sets a maximum to the number of filtered events to be considered. Again function *tl* can be used. Assuming a maximum m, the result of this third abstraction is $\sigma_3 = tl^m(\sigma_2)$. The fourth abstraction is based on q. Recall that there are three possible values: $q = seq$ (sequence), $q = ms$ (multiset), or $q = set$ (set). Hence, we take the sequence σ_3 resulting from the first three abstractions and use σ_3 (no abstraction), $par(\sigma_3)$ (i.e., construct a multi-set and remove the ordering) or $set(par(\sigma_3))$ (i.e., construct a set and remove both ordering and frequencies).

Now we can formalize examples of *state*() functions. For example, consider Figure 7(a) where $h = \infty$, $F = A$, $m = \infty$, $q = set$. In this case, $state(\sigma, k) = set(par(tl^{\infty}(tl^{\infty}(hd^k(\sigma)) \uparrow A)))$. This can be simplified to $state(\sigma, k) = set(par(hd^k(\sigma) \uparrow A))$. In Figure 7(b), where $h = \infty$, $F = \{A, D, E\}$, $m = 1$, and $q = seq$, the function is $state(\sigma, k) = tl^1(tl^{\infty}(hd^k(\sigma)) \uparrow \{A, D, E\})$. Using these two *state*() functions and the corresponding V values, the two transition systems shown in Figure 7 can be obtained by simply applying Definition 3.

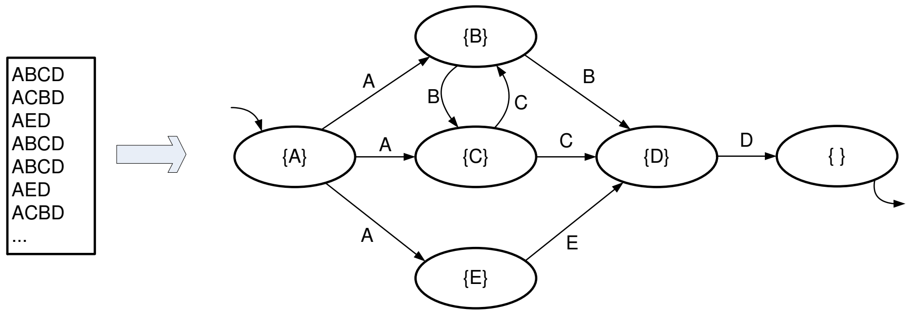

Fig. 8. A transition system constructed based on the future of a case (postfix) with abstractions $h = \infty$, $F = A$ (i.e., all activities), $m = 1$, $q = set$, and $V = A$.

The examples so far have focused on the past of a case (i.e., prefixes). A similar approach can be used for postfixes (i.e., future). In this situation $\sigma_0 = tl^{n-k}(\sigma)$ is the complete postfix of some case σ of length n after k steps. The first abstraction presented in Section 3.2 can be tackled using function *hd*. Assuming a horizon h, is results in $\sigma_1 = hd^h(\sigma_0)$. Assuming a filter F, the result of the second abstraction is $\sigma_2 = \sigma_1 \uparrow F$. The third abstraction sets a maximum to the number of filtered events: $\sigma_3 = hd^m(\sigma_2)$ The fourth abstraction take is identical to using a prefix, i.e., σ_3, $par(\sigma_3)$ or $set(par(\sigma_3))$. Figure 8 shows an abstraction based on the postfix and $m = 1$ (i.e., at most one filtered event is considered).

If both the past and future are used, then for both prefix and postfix an abstraction needs to be selected and the state is then determined by pairing both abstractions. For example, $state(\sigma, k) = (par(tl^{\infty}(tl^{2}(hd^{k}(\sigma)) \uparrow \{A, B\})),$ $set(par(hd^{2}(hd^{\infty}(tl^{n-k}(\sigma)) \uparrow \{B, C, D\}))))$.

3.4 Extensions

We have now introduced and formalized the basic approach to construct a transition system based on an event log. The next step is to transform this transition system into a process model. However, before discussing the second step, we first discuss two types of extensions of the basic approach. To avoid an overkill of notations, we only present these extensions informally.

Massaging the transition system The first type of extensions is related to "massaging" the transition system after it is generated. This is intended to "pave the path" for the second step. For example, one may remove all "self-loops", i.e., transitions of the form $s \xrightarrow{a} s$ (cf. Figure 9(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", i.e., 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 (cf. Figure 9(b)). 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 $s_3 \xrightarrow{a_1} s_4$.

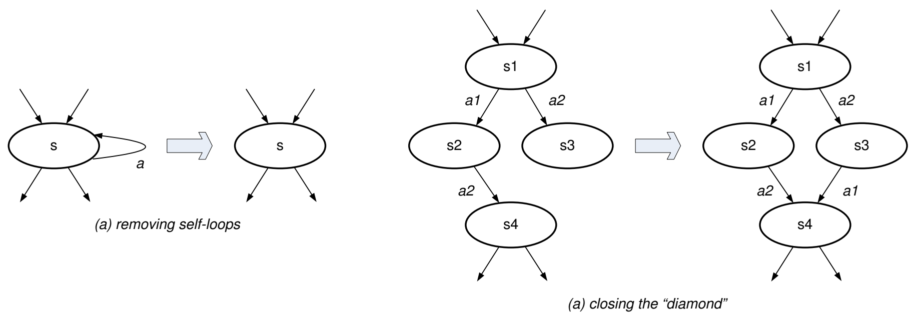

Fig. 9. Two examples of modifications of the transition system to aid the construction of the process model.

Incorporating other perspectives The second type of extensions is related to the input, i.e., the "richness" of the event log. In Definition 1, a simple log was assumed, i.e., a case is described as a sequence of activities and a log is a set of such simple sequences. In reality, one knows much more about events. Most information systems do not just record the ordering of activities but also timestamps and information about resources, data, transactional information, etc.

Definition 5 (Trace, Event log). *Let E be a set of events. Based on E there is a set of p* properties*: $\{prop_1, \ldots prop_p\}$. Each property is a function with a particular range, i.e., for $1 \leq i \leq p$: $prop_i \in E \rightarrow R_i$. Given an event $e \in E$, $prop_i(e)$ maps the event onto a particular property of the event, e.g., its timestamp, the activity executed, the person executing the event, etc. Based on E and the set of properties, we define $\sigma \in E^*$ as a (complex)* trace *and $L \in \mathcal{P}(E^*)$ as a (complex)* event log*.*

Note that Definition 1 can be seen as a special case of the above definition with only one property, being the activity itself. Some examples of typical property functions are:

- $\mathit{activity} \in E \rightarrow A$ where A is the set of activities. $\mathit{activity}(e)$ is the activity that e refers to.
- $\mathit{timestamp} \in E \rightarrow TS$ where TS is the set of timestamps. $\mathit{timestamp}(e)$ is the time that e occurred.
- $\mathit{performer} \in E \rightarrow P$ where P is the set of persons. $\mathit{performer}(e)$ is the person executing e.
- $\mathit{trans\_type} \in E \rightarrow \{\mathit{enable}, \mathit{start}, \mathit{complete}, \mathit{abort}, \ldots\}$. $\mathit{trans\_type}(e)$ is the type of transaction, e.g., if $\mathit{activity}(e) = \mathit{conduct\_interview}$ and $\mathit{trans\_type}(e) = \mathit{start}$, then e is the start of the interview.

There may also be property functions describing data attributes of an event or linking events to business objects.

For convenience, we assume that all property functions are extended to sequences, i.e., if $\sigma = \langle e_1, e_2, \ldots, e_n \rangle \in E^*$, then $prop_i(\sigma) = \langle prop_i(e_1), prop_i(e_2), \ldots, prop_i(e_n) \rangle \in R_i^*$.

The goal of the additional information captured in events is to provide for more ways of extracting transition systems. One way would be to allow for state functions of the form $\mathit{state} \in (E^* \times \mathbb{N}) \not\rightarrow R$, i.e., defining dedicated abstractions based on the various properties of a case. Another approach would be to use the abstractions defined earlier (i.e., h, F, m, q, and V) and first project the complex trace onto a simple trace. For example, transform the complex log $L \in \mathcal{P}(E^*)$ into a simple log $L' = \{\mathit{activity}(\sigma) \mid \sigma \in L\}$ by projecting each event onto its activity. In a similar way $L' = \{\mathit{performer}(\sigma) \mid \sigma \in L\}$ could be used to explore the transfer of work from one person to another using the abstractions defined earlier. In our implementation described in Section 5, we will show that we essentially used the second approach. Independent of the approach chosen, this will result in a $\mathit{state}()$ function that can be used to construct a transition system as defined in Definition 3.

4 Synthesis using Regions (Step 2)

In this section, we present the second step of our approach. In this second step, a process model is synthesized from the transition system resulting from the first step. In this paper and our implementation, we use the well-known "theory of regions" [15, 18, 23] to construct a Petri net.

4.1 Constructing Petri Nets Using Regions

The *synthesis problem* is the problem to construct, for a given behavioral specification, a Petri net such that the behavior of this net coincides with the specified behavior (if such a net exists). There are basically two approaches to tackle this problem. Both use the notion of "regions". The *state-based region theory* [15, 18, 23] 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. The *language-based region theory* [11, 29, 30, 40] does not take a transition system as input but a language (e.g., a regular language or simply a finite set of sequences, i.e., a log). However, the basic principe is similar.

Given the fact that, in the first step, we construct, in a controlled manner, a transition system, it seems most natural to use the state-based region theory. There are many variants and extensions of this theory. However, to present the basic idea we start with the classical definition of a *region.*

Definition 6 (Region). *Let $TS = (S, E, T)$ be a transition system and $S' \subseteq S$ be a subset of states. S' is a* region *if for each event $e \in E$ one of the following conditions hold:*

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 $s_1, s_2 \notin S'$*

Figure 10 illustrates this notion and puts the idea in the context of this paper. The log is converted into a transition system as shown earlier. The transition system consists of 8 states. The set consisting of just state $\{\}$ is a region because all A-transitions leave this region and all other transitions never cross this region. The set consisting of states $\{A\}$ and $\{A, B\}$ is a region because all A-transitions enter this region, all C-transitions leave this region, all E-transitions leave this region, and all other transitions (i.e., B and D) never cross this region. Figure 10 also shows a region consisting of three states: $\{A, B\}$, $\{A, B, C\}$, and $\{A, E\}$. Note that this region is shown by *two* connected areas in Figure 10. All B-transitions enter this region, all E-transitions enter this region, all D-transitions leave this region, and all other transitions never cross this region. As Figure 10 shows regions correspond to places in the Petri net. The transitions that enter a region r are input transitions of the corresponding r place. Transitions that

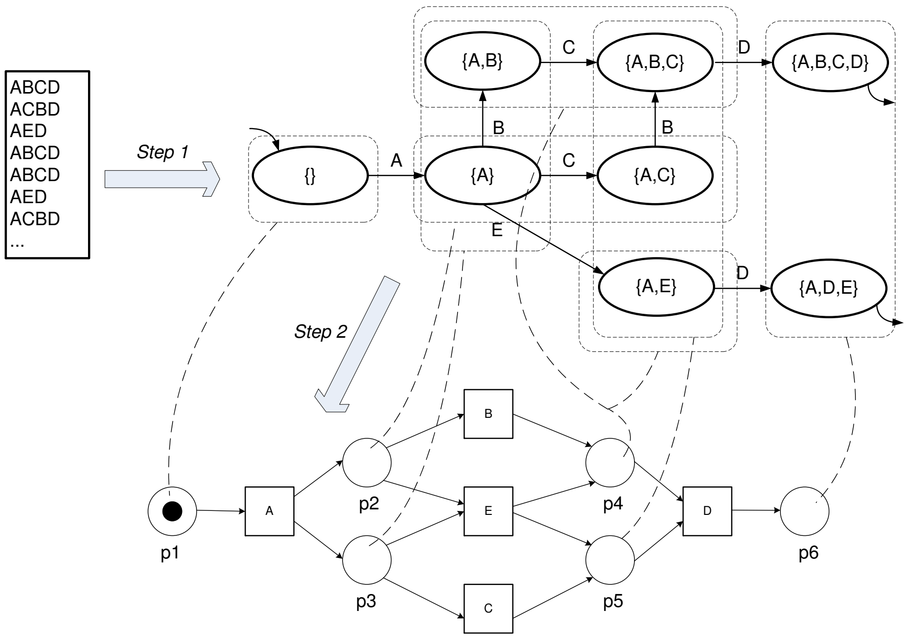

Fig. 10. A transition system based on prefixes and abstractions $h = \infty$, $F = A$, $m = \infty$, $q = set$, and $V = A$, is converted into a Petri net using the "theory of regions". The six regions correspond to places in the Petri net.

leave region r are output transitions of r. The transitions that do not cross r are not connected to the corresponding place.

Any transition system $TS = (S, E, T)$ has two trivial regions: $\emptyset$ (the empty region) and S (the region consisting of all states). Since these hold no information about the process, 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 non-trivial region r' which is a subregion of r. Note that all regions in Figure 10 are minimal. Region r is a *preregion* of e if there is a transition labeled with e which 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. An example of a Petri net synthesized from a transition system using this simple algorithm is given in Figure 10.

The first papers on the "theory of regions" only dealt with a special class of transition systems called *elementary transition systems*. See [9,10,18] 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. [14,15], 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. For this approach it has been shown that the *reachability graph* of the synthesized Petri net is *bisimilar* to the initial transition system even if the transition system is non-elementary.

To illustrate the problem of non-elementary transition systems, consider Figure 11. The transition system is obtained by abstracting away B and C, e.g., use $h = \infty$, $F = \{A, D, E\}$, $m = 1$, $q = seq$, and $V = \{A, D, E\}$ like in Figure 7 and then remove the self loop. An isomorphic transition system can be obtained by filtering out B and C first and then use prefixes of length 1 without any further abstractions. This transition system is not elementary. The problem is that there are two states $\langle A \rangle$ and $\langle E \rangle$ 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 11 fails to construct a bisimilar Petri net. However, using label-splitting as presented in [14,15], 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 [14]. As shown in [14,15] it is always possible to construct an equivalent Petri net. However, label-splitting may lead to larger Petri nets.

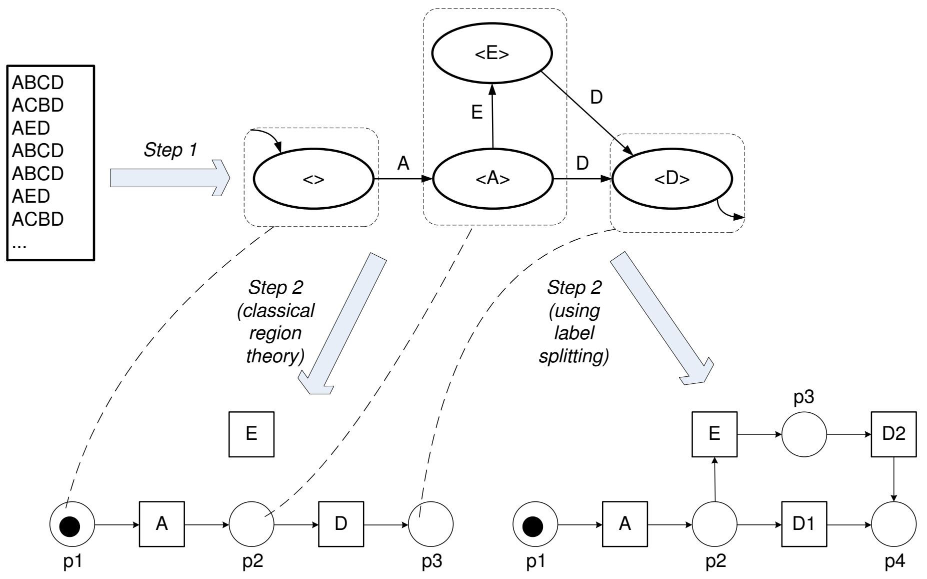

Fig. 11. The transition system generated based on the log is not elementary. Therefore, the generated Petri net using classical region theory is not equivalent (modulo bisimilarity). However, using "label-splitting" an equivalent Petri net can be obtained.

4.2 More on Regions

In this paper, we do not propose a new approach to construct regions. However, we would like to reflect on the application of region theory in the context of mining.

Generalization Region theory aims at synthesis, i.e., the Petri net should have a behavior which is identical to the initial behavior specified. Therefore, the Petri net shown on the left-hand side of Figure 11 is considered to be a problem from the synthesis point of view. This is the reason why label-splitting is used to construct a bisimilar Petri net if the net is not elementary, cf. right-hand side of Figure 11.

In this paper, we have used an approach where all generalizations take place in the first step. This way the purpose of each step is clear: (a) *Step 1 is concerned with selecting the desired behavior* (i.e., abstraction and generalization), and (b) *Step 2 is only concerned with generating the corresponding Petri net representation.* This means that the analyst can influence the first step but not the second. Note that the Petri net shown on the left-hand side of Figure 11 is *generalizing things in an uncontrolled manner.* Because of representation issues E is suddenly allowed to execute an arbitrary number of times while in the log it was never executed multiple times. This is undesirable and shows that region theory tends to yield overfitting models but at the same time it may generalize things in an uncontrolled manner, i.e., the generalization is driven by representational issues rather than behavioral ones.

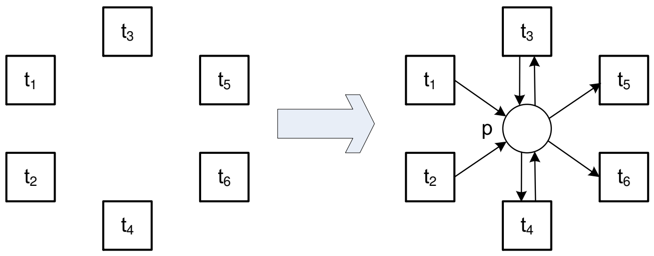

Fig. 12. The left-hand side Petri net can "parse" any log while the addition of place p on the right-hand side imposes a constraint on the traces that can be parsed.

Although in this paper and the implementation described in Section 5 we only generalize in the first step, it is interesting to note that existing synthesis approaches can be fine-tuned for mining [11,40]. Such approaches are completely different from the main approach described in this paper and consist of only one step that constructs a Petri net while generalizing at the same time.

Figure 12 shows the basic idea. Suppose one has a log with activities $A = \{t_1, \ldots, t_6\}$. It is easy to see that the Petri net on the left-hand side in Figure 12 can "parse" this log, i.e., the model fits any log containing activities A. The addition of a place can be seen as adding a constraint. By adding place p in Figure 12 a constraint is introduced. Suppose that $\sigma \in L$. Let σ_1 and $\sigma_2 = \sigma_1\langle t'\rangle$ (i.e., σ_1 concatenated with some t'), be two prefixes of σ. Assume that p has initially k tokens. Then p can only be added if $k + par(\sigma_1)(t_1) + par(\sigma_1)(t_2) + par(\sigma_1)(t_3) + par(\sigma_1)(t_4) - par(\sigma_2)(t_3) - par(\sigma_2)(t_4) - par(\sigma_2)(t_5) - par(\sigma_2)(t_6) \geq 0$.[8] Hence, we can add any place such that above constraint holds for any $\sigma \in L$. In [11, 29, 30, 40] several approaches are presented to add as few places as possible while maximally constraining the net (without violating the constraints mentioned above). It is obvious that any cost function can be associated to the addition of a place. For example, there may a "penalty" for places that are not easy to understand. Such penalties can easily be combined with constraints into an integer lineair programming problem [40]. This way regions can be used directly to balance between overfitting and underfitting.

Selecting the target format 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.

[8] Recall that $par(\sigma_1)(t)$ is the number of times that t occurred in σ_1.

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 [14,15] 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. Figure 13 shows the Petri net obtained by using the approach in [14, 15] while demanding the result to be free-choice. Note that in order to make the net free-choice the labels B and C were split.

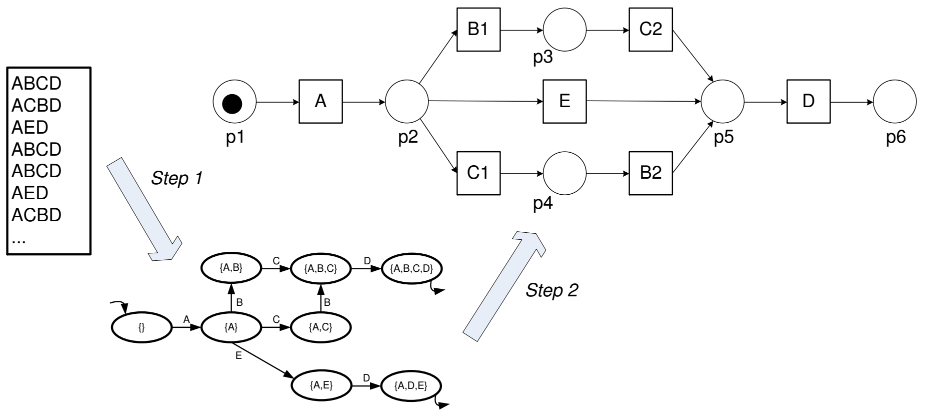

Fig. 13. The techniques presented in [14, 15] can be used to obtain a free-choice Petri net. Note that labels B and C are split in order to achieve this.

The Petri net in Figure 13 is bisimilar to the transition system and the set of possible traces of the model coincides with the initial log. However, other approaches such as the one presented [40] do not aim at a model whose possible traces of the model coincides with the initial log. In such an approach it is possible to allow only the adding of places that have the desired properties. One can use cost functions to decide on the addition of a place such as the one shown in Figure 12. Using such an approach it is also possible to enforce that the model is e.g. free-choice.

It is interesting to think about “cost models” in the context of process mining to balance overfitting and underfitting. It may be that there is a simple model that almost captures the behavior observed while all models that exactly capture the behavior are much more complex. In such a case, it may be wise to show the simple model to the user. The addition of a place has a “cost” in terms of adding to the complexity of the result and potentially overfitting. Note that a place with many input and output transitions is typically a sign of overfitting.

However, not adding a place has a "cost" in terms of underfitting, i.e., allow for too much behavior not present in the log. The notion of costs is also interesting when dealing with noise (cf. Section 2.3), i.e., a place may be added because the majority of the cases suggest that this is natural while a small fraction of the log cannot be reproduced because of this place.

5 Implementation and Evaluation

In the previous two sections, we presented a two-step approach. In Section 3, we showed how transition systems can be extracted in a controlled way while balancing between overfitting and underfitting. Section 4 showed that region theory can be used to convert the result into an equivalent Petri net, i.e., concurrency is detected and moved to the net level. The ideas presented in this paper have been fully implemented in the context of *ProM*.[9] The ProM serves as a testbed for our process mining research [2] and can be downloaded from `www.processmining.org`. Starting point for ProM is the MXML format. This is a vendor-independent format to store event logs. Similar to Definition 5, this format allows for the recording of timestamps, data elements, performers, etc.

Note that in this paper we focused on discovering Petri nets and not models in other notations. However, the resulting Petri net can then be converted into the desired notation, e.g., BPMN, EPCs, UML activity diagrams, etc. This is standard functionality of ProM, therefore, we do not elaborate on this.

5.1 ProMimport

The ProMimport Framework [25] allows developers to quickly implement plug-ins that can be used to extract information from a variety of systems and convert it into the MXML format (cf. `promimport.sourceforge.net`). There are standard import plug-ins for a wide variety of systems, e.g., workflow management systems like Staffware, case handling systems like FLOWer, ERP components like PeopleSoft Financials, simulation tools like ARIS and CPN Tools, middleware systems like WebSphere, BI tools like ARIS PPM, etc. Moreover, it has been used to develop many organization/system-specific conversions (e.g., hospitals, banks, governments, high-tech systems, etc.).

5.2 ProM

Once the logs are converted to MXML, ProM can be used to extract a variety of models from these logs. ProM provides an environment to easily add so-called "plug-ins" that implement a specific mining approach. Although the most interesting plug-ins in the context of this paper are the mining plug-ins, it is important to note that there are in total five types of plug-ins:

[9] Note that the described functionality is present in the so-called "Nightly Builds" of ProM and not yet in the released version (4.2). These Nightly Builds can also be downloaded via `www.processmining.org` and all functionality will be present in Release 5.0 of ProM.

Mining plug-ins which implement some mining algorithm, e.g., mining algorithms that construct a Petri net based on some event log, or algorithms that construct a transition system or a social network from an event log.

Export plug-ins which implement some "save as" functionality for some objects (such as graphs). For example, there are plug-ins to save EPCs, Petri nets, spreadsheets, etc.

Import plug-ins which implement an "open" functionality for exported objects, e.g., load Petri nets that are generated by Petrify or EPCs from ARIS.

Analysis plug-ins which typically implement some property analysis on some mining result. For example, for Petri nets there is a plug-in which constructs place invariants, transition invariants, and a coverability graph.

Conversion plug-ins which implement conversions between different data formats, e.g., from EPCs to Petri nets and from Petri nets to YAWL and BPEL.

Currently, there are more than 230 plug-ins [2]. One of these plug-ins is the mining plug-in that generates the transition system that can be used to build a Petri net model. Another plug-in is the conversion plug-in that uses Petrify to synthesize a Petri net. Petrify is embedded in ProM and uses the algorithms developed by Cortadella et al. [14, 15].

Figure 14 shows the mining and conversion plug-in in ProM. Although the notation is slightly different, it is easy to see that the results of the two steps are indeed as shown earlier in Figure 10. What is more difficult to see is that the top window allows for all kinds of abstractions. The tabs *Model element*, *Originator*, *Event type*, and *Attributes* refer to the type of information to be used. This corresponds to the selection of properties used for building the transition system (cf. the property functions mentioned in Definition 5). Note that any conjunction of these properties can be used, e.g., a state may be based on the person that executed the last task and the name of the next activity. Moreover, all abstractions mentioned in this paper are available. In Figure 14 the following settings are used: prefixes with $h = \infty$, $F = A$, $m = \infty$, and $q = set$. Under tab *Visible* the transition names to be shown one the arcs in the diagram can be selected (i.e., the value of V). The *Modifications* tab allows for the extensions mentioned in Section 3.4.

As Figure 14 illustrates, the ideas presented in this paper have been fully implemented.

5.3 Evaluation

In the introduction we mentioned that existing process mining approaches suffer from the problems such as the inability to deal with advanced control-flow constructs (e.g., non-local dependencies, skips, duplicates, etc.), the inability to guarantee the correctness of the model (e.g., absence of deadlocks, etc.), and inability to balance between overfitting and underfitting. To illustrate that the approach presented here can overcome these problems, we use a slightly larger example.

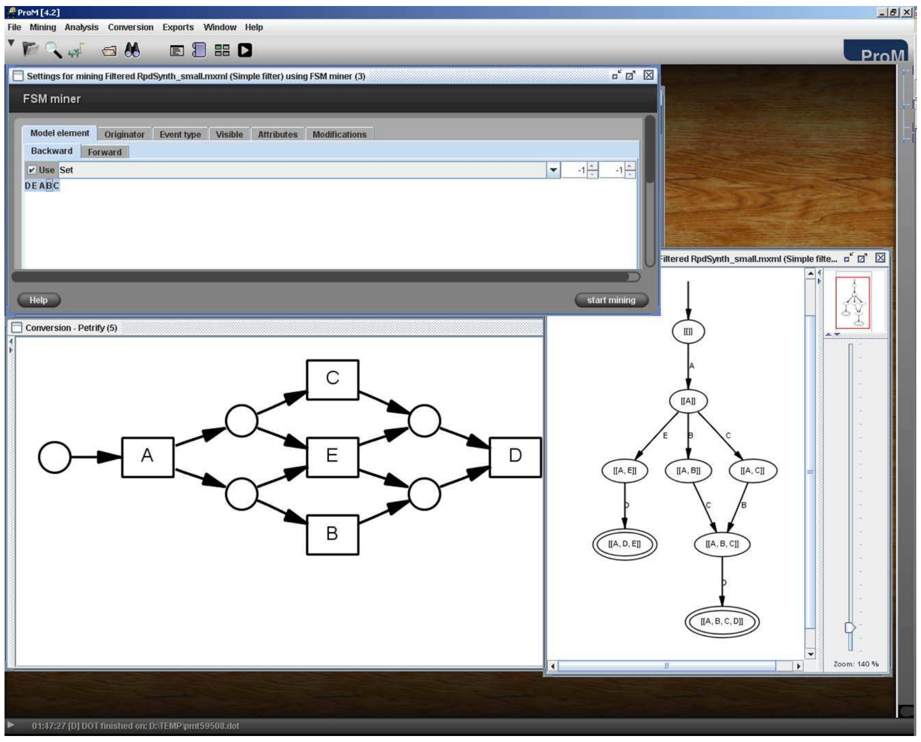
ProM [4.2]
File Mining Analysis Conversion Exports Window Help
Settings for mining Filtered RpdSynth_small.mxml (Simple filter) using FSM miner (3)
FSM miner
Model element Originator Event type Visible Attributes Modifications
Backward Forward
Use Set -1 -1
D E A B C
Help
start mining
Conversion - Petrify (5)
A
C
E
B
D
Filtered RpdSynth_small.mxml (Simple filte...
[[]]
A
[[A]]
E
B
C
[[A, E]]
[[A, B]]
[[A, C]]
D
C
B
[[A, D, E]]
[[A, B, C]]
D
[[A, B, C, D]]
Zoom: 140 %
01:47:27 [D] DOT finished on: D:\TEMP\pmt59508.dot

Fig. 14. A screenshot of ProM while analyzing the running example. The top window shows the various abstractions that can be selected. The right window show the result of Step 1 and the left window shows the result of Step 2.

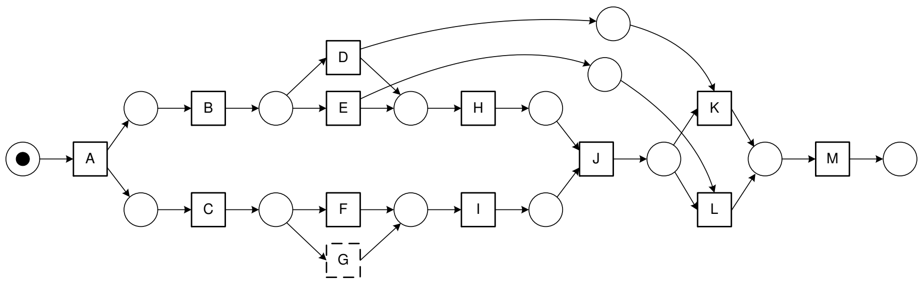

Fig. 15. Thousand cases have been generated according to this process model. Complications are that G is invisible (see dashed transition) and that there is a non-local dependency controlling the choice between K and L.

Figure 15 shows a process modeled in terms of a Petri net. Note that this model has two types of constructs that most process mining algorithms have problems dealing with. The first construct is activity G that is not being logged. As a result F can be skipped. Since such a skip is not recorded, it is not trivial to detect it. The second construct is the non-local dependency controlling the choice between K and L. Note that K (L) is only selected if also D (E) was selected. However, there are always at least two activities in-between D (E) and K (L) (activities H and J). Most process mining techniques have problems with at least one of these two constructs. Based on the model in Figure 15, we have randomly generated 1000 cases using the simulation tool CPN Tools. The event log of CPN Tools is converted into MXML such that we can apply a wide variety of process mining algorithms (including the one presented in this paper).

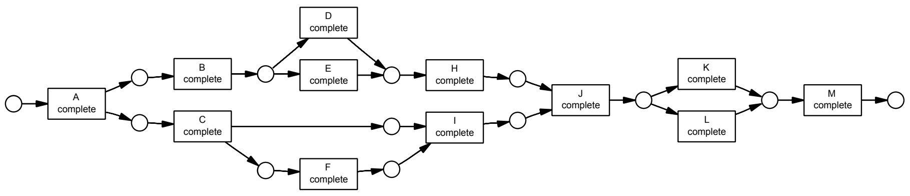

Fig. 16. The Petri net discovered by the α algorithm in ProM. Note that although F can be skipped in reality this is not possible in the model. Moreover, the non-local dependencies are not discovered.

The 1000 cases have been generated randomly. The model allows for 80 different traces (assuming G is visible). However, *only 66 of these traces actually occur in the log*. Moreover, some traces appear frequent in the log (e.g., there is a trace that is repeated 83 times in the log) while others are unique. This nicely illustrates the discussion on completeness in Section 2.3.

Figure 16 shows the result when applying the α algorithm [7]. Note that this model and its layout are automatically generated from the log without human intervention. (Note that the ProM plug-in constructing the Petri net using the α algorithm inserts transactional information. Therefore, each transition is classified as a "complete transition" in Figure 16.) As can be seen, the model is incorrect because it does not allow for the skipping of F. This is caused by the fact that G is invisible. Moreover, there are no connections from the choice between D and E to the choice between K and L.

Some of the algorithms presented in literature can overcome these problems. For example, the α^{++} algorithm presented in [42] and implemented in ProM can discover the non-local dependencies between D (E) and K (L). However, it cannot handle the skipping of F leading to the same problem as shown in Figure 16. The multi-phase miner [20] (also implemented in ProM) can handle the skipping of F but not the non-local dependencies. The genetic miner [3, 31] may be able to discover both types of constructs correctly. However, this takes a lot of computation time and the end result is not guaranteed. Older algorithms such as [12] are not able to discover concurrency. Therefore, Figure 15 is a nice benchmark example to evaluate our two-step approach.

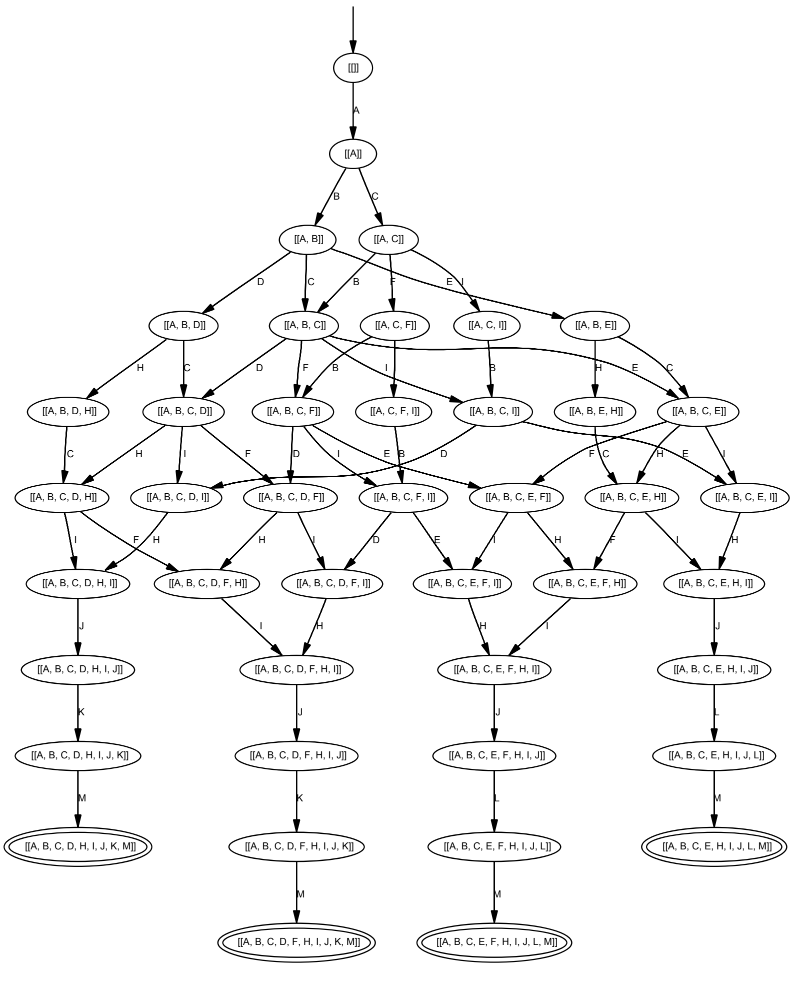

Fig. 17. The transition system constructed by ProM.

Figure 17 shows the transition system based the log containing 1000 cases and function $state(\sigma, k) = set(par(tl^{\infty}(tl^{\infty}(hd^{k}(\sigma)) \uparrow A))) = set(par(hd^{k}(\sigma)))$, i.e., we use prefixes and the abstractions $h = \infty$, $F = A$, $m = \infty$, $q = set$, and $V = A$. The transition system in Figure 17 can easily be converted into the Petri net shown in Figure 18 using the "theory of regions". Note that the Petri net and its layout are automatically generated using ProM.

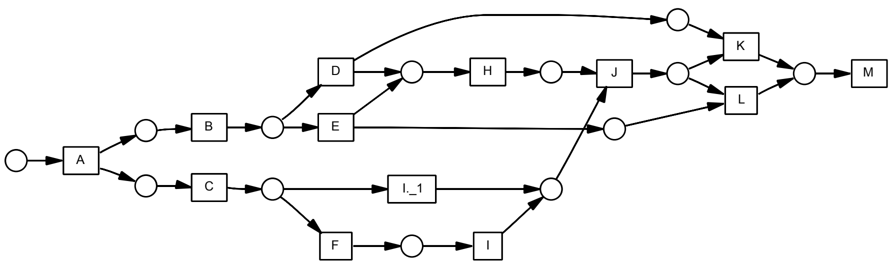

Fig. 18. The Petri net constructed by ProM based on the transition system of Figure 17.

Figure 18 captures the non-local dependencies and the skipping of F correctly. The two places between D and E and K and L correctly model the dependencies present. The skipping is modeled by label-splitting, i.e., there are two transition referring to activity I: I_1 and I. Note that the notation generated by ProM is slightly different from the notation used in this paper. However, the correspondence should be clear. Also note that the output place of M is suppressed. However, this is merely a technicality and not a limitation.

The above example shows that the two-step approach is able to discover complex constructs and is capabele of delivering models that are guaranteed to satisfy various correctness criteria. The reason is that the transition system is a *controlled* abstraction of the log and has guaranteed properties that are preserved through the construction of the Petri net. (Recall that the Petri net is bisimilar.)

Another property of the two-step approach is that it is configurable, i.e., by changing settings different models can be constructed depending on the desired abstraction/generalization. This way one can balance between overfitting and underfitting in a controlled way. Figure 19 shows an alternative process model constructed using the same log but a different abstraction. In this case, we again use prefixes and $h = \infty$, $m = \infty$, $q = set$, and $V = A$. However, now $F = \{A, B, D, E, G, H, J, K, L, M\}$. This means that we tell the algorithm not to use C, F, and I for constructing the state but to still include the activities in the process model. As Figure 19 shows C, F, and I can be executed in-between A and J. However, because these activities are not used in the state information, the process model only indicates that they can be executed without indicating an order or frequency.

Figure 20 shows yet another alternative process model. This example has been added to show the use of postfixes and a limited horizon. In this case,

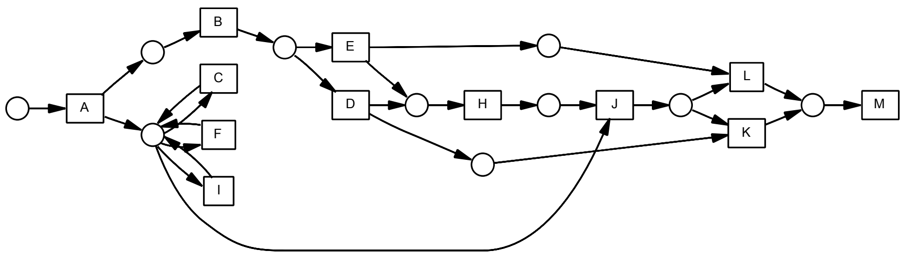

Fig. 19. Another Petri net constructed using the abstraction $F = \{A, B, D, E, G, H, J, K, L, M\}$.

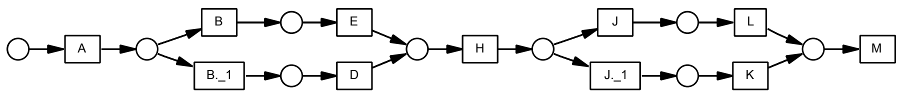

Fig. 20. A Petri net constructed by filtering the log and building states based on postfixes using abstraction $m = 1$.

$h = \infty$, $F = A$, $m = 1$, $q = set$, and $V = A$. Moreover, activities C, F, and I have now been filtered out of the log before applying the algorithm. Hence, they do not appear in the model at all (i.e., also not on the arcs like in Figure 19). It is interesting to note that the non-local dependency is *deliberately* abstracted from by setting $m = 1$. Moreover, since the state is based on the future (i.e., next activity), activities B and J are split. The reason is that using this state function, the next step is already known after executing B or J. Therefore, the moment of choice is moved to an earlier point.

Figures 18, 19, and 20 show that based on the same log different models can be constructed based on the desired abstraction/generalization. As far as we know, this is the only algorithm described in literature able to balance between overfitting and underfitting in a controlled way.

6 Related Work

Since the mid-nineties several groups have been working on techniques for process mining [4, 7, 8, 12, 16, 20, 21, 41], 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 [8]. In parallel, Datta [16] looked at the discovery of business process models. Cook et al. investigated similar issues in the context of software engineering processes [12]. Herbst [26] 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 [7] 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 [20, 21] a more robust but less precise approach is presented.

In this paper we do not consider issues such as noise (cf. Section 2.3). Heuristics [41] or genetic algorithms [3,31] have been proposed to deal with issues such as noise. It appears that some of the ideas presented in [41] can be combined with other approaches, including the one presented in this paper.

The second step in our approach uses the "theory of regions". In our approach we use the so-called state-based regions as defined in [9,10,14,15,23]. This way, transition systems can be mapped onto Petri nets using synthesis. Initially, the theory could be applied only to a restricted set of transition systems. However, over time the approach has been extended to allow for the synthesis from any finite transition system. In this paper, we use *Petrify* [13] for this purpose. The idea to use regions has been mentioned in several papers. However, only recently people have been applying state-based regions to process mining [28]. It is important to note that the focus of regions has been on the synthesis of models exactly reproducing the observed behavior (i.e., the transition system). An important difference with our work is that we try to generalize and deduce models that allow for more behavior, i.e., our approach supports the balancing between "overfitting" and "underfitting". In our view, this is the most important challenge in process mining research.

Recently, some work on language-based regions theory appeared [11, 29, 30, 40]. In [11, 40] it is shown how this can be applied to process mining. These approaches are very interesting and directly construct a Petri net. They are not building an intermediate transition system. This has advantages, e.g., in terms of efficiency, but also disadvantages because the approach is less configurable.

Process mining can be seen in the broader context of Business Process Intelligence (BPI) and Business Activity Monitoring (BAM). In [24, 38] a BPI toolset on top of HP's Process Manager is described. The BPI toolset includes a so-called "BPI Process Mining Engine". In [33] Zur Muehlen describes the PISA tool which can be used to extract performance metrics from workflow logs. Similar diagnostics are provided by the ARIS Process Performance Manager (PPM) [27]. The tool is commercially available and a customized version of PPM is the Staffware Process Monitor (SPM) [39] which is tailored towards mining Staffware logs. It should be noted that BPI tools typically do not allow for process discovery and offer relatively simple performance analysis tools that depend on a correct a-priori process model. One of the few commercial tools that supports process mining is the BPM|suite of Pallas Athena. This tool is using the ideas behind ProM and integrates this into a BPM product.

An earlier version of this paper appeared as a technical report [5]. Here additional examples are shown and the role of data/documents for state construction is discussed in more detail.

7 Conclusion

This paper presented a new two-step process mining approach. It uses innovative ways of constructing transition systems and regions to synthesize process models in terms of Petri nets. Using this approach, it is possible to discover process models that adequately describe the behavior recorded in event logs. These logs may come from a variety of information systems e.g., systems constructed using ERP, WFM, CRM, SCM, and PDM software. The application is not limited to repetitive administrative processes and can also be applied to development processes and processes in complicated professional/embedded systems. Moreover, process mining is suitable for the monitoring of interacting web services.

Existing approaches typically provide a single process mining algorithm, i.e., they assume "one size fits all" and cannot be tailored towards a specific application. Unlike existing approaches, we allow for a wide variety of strategies, and, as a side-effect, overcome some of the problems of existing approaches. Selecting the right state representation aids in balancing between "overfitting" (i.e., the model is over-specific and only allows for the behavior that happened to be in the log) and "underfitting" (i.e., the model is too general and allows for unlikely behavior). The relation between the log and the transition system is much more direct than in existing approaches. Therefore, it is easier to control the abstraction/generalization and it is clear which properties are preserved. For the transformation of a transition system into a Petri net we use regions. The idea is that, in this second step, the behavior is not changed, i.e., using regions we look for concurrency and move this to the net level. As a result the model becomes more compact and can easily be translated to other process modeling languages (EPCs, BPMN, UML activity diagrams, etc.).

The approach has been fully implemented in ProM and the resulting process mining tool can be downloaded from `www.processmining.org`.

Future work is aiming at a better support for strategy selection and new synthesis methods. The fact that our two-step approach allows for a variety of strategies makes it very important to support the user in selecting suitable strategies depending on the characteristics of the log and the desired end-result. Practical experiments point out the need for better synthesis methods. Existing region-based approaches implemented in tools such as Petrify have severe performance problems and typically lead to less intuitive models. The "theory of regions" aims at developing an equivalent Petri net while in process mining a simple less accurate model is more desirable than a complex model that is only able to reproduce the log. Hence it is interesting to investigate "new theories of regions" tailored towards process mining. Some initial work in this direction has already been presented in [11, 40].

8 Acknowledgements

This research is supported by EIT, NWO-EW, and the Technology Foundation STW. Moreover, we would like to thank the many people involved in the development of ProM.

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, 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.
3. W.M.P. van der Aalst, A.K. Alves de Medeiros, and A.J.M.M. Weijters. Genetic Process Mining. In G. Ciardo and P. Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 48–69. Springer-Verlag, Berlin, 2005.
4. 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.
5. W.M.P. van der Aalst, V. Rubin, B.F. van Dongen, E. Kindler, and C.W. Günther. Process Mining: A Two-Step Approach using Transition Systems and Regions. BPM Center Report BPM-06-30, BPMcenter.org, 2006.
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, 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.
8. R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In *Sixth International Conference on Extending Database Technology*, pages 469–483, 1998.
9. 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.
10. 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.
11. 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.
12. 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.

13. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.
14. 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.
15. 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.
16. A. Datta. Automating the Discovery of As-Is Business Process Models: Probabilistic and Algorithmic Approaches. *Information Systems Research*, 9(3):275–301, 1998.
17. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.
18. J. Desel and W. Reisig. The Synthesis Problem of Petri Nets. *Acta Informatica*, 33(4):297–315, 1996.
19. J. Desel, W. Reisig, and G. Rozenberg, editors. *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2004.
20. 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.
21. 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.
22. M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology.* Wiley & Sons, 2005.
23. A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures - Part 1 and Part 2. *Acta Informatica*, 27(4):315–368, 1989.
24. D. Grigori, F. Casati, M. Castellanos, U. Dayal, M. Sayal, and M.C. Shan. Business Process Intelligence. *Computers in Industry*, 53(3):321–343, 2004.
25. C. Günther and W.M.P. van der Aalst. A Generic Import Framework for Process Event Logs. In J. Eder and S. Dustdar, editors, *Business Process Management Workshops, Workshop on Business Process Intelligence (BPI 2006)*, volume 4103 of *Lecture Notes in Computer Science*, pages 81–92. Springer-Verlag, Berlin, 2006.
26. 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.
27. IDS Scheer. ARIS Process Performance Manager (ARIS PPM): Measure, Analyze and Optimize Your Business Process Performance (whitepaper). IDS Scheer, Saarbruecken, Gemany, http://www.ids-scheer.com, 2002.
28. E. Kindler, V. Rubin, and W. Schäfer. Process Mining and Petri Net Synthesis. In J. Eder and S. Dustdar, editors, *Business Process Management Workshops*, volume

4103 of *Lecture Notes in Computer Science*, pages 105–116. Springer-Verlag, Berlin, September 2006.

29. R. Lorenz, R. Bergenthum, J. Desel, and S. Mauser. Synthesis of Petri Nets from Finite Partial Languages. In T. Basten, G. Juhás, and S.K. Shukla, editors, *International Conference on Application of Concurrency to System Design (ACSD 2007)*, pages 157–166. IEEE Computer Society, 2007.
30. 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.
31. A.K.A. de Medeiros. *Genetic Process Mining.* PhD thesis, Eindhoven University of Technology, Eindhoven, 2006.
32. A.K.A. 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.
33. M. zur Mühlen and M. Rosemann. Workflow-based Process Monitoring and Controlling - Technical and Organizational Issues. In R. Sprague, editor, *Proceedings of the 33rd Hawaii International Conference on System Science (HICSS-33)*, pages 1–10. IEEE Computer Society Press, Los Alamitos, California, 2000.
34. 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.
35. A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In C. Bussler et al., editor, *BPM 2005 Workshops (Workshop on Business Process Intelligence)*, volume 3812 of *Lecture Notes in Computer Science*, pages 163–176. Springer-Verlag, Berlin, 2006.
36. A. Rozinat and W.M.P. van der Aalst. Decision Mining in ProM. In S. Dustdar, J.L. Faideiro, and A. Sheth, editors, *International Conference on Business Process Management (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 420–425. Springer-Verlag, Berlin, 2006.
37. 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.
38. M. Sayal, F. Casati, U. Dayal, and M.C. Shan. Business Process Cockpit. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 880–883. Morgan Kaufmann, 2002.
39. TIBCO. TIBCO Staffware Process Monitor (SPM). http://www.tibco.com, 2005.
40. J.M.E.M. van der Werf, B.F. van Dongen, C.A.J. Hurkens, , and A. Serebrenik. Process Discovery using Integer Linear Programming. Computer Science Report (08-04), Eindhoven University of Technology, Eindhoven, The Netherlands, 2008.
41. 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.
42. 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.