Process Mining: A Two-Step Approach using Transition Systems and Regions

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

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
{w.m.p.v.d.aalst,b.f.v.dongen,c.w.gunther}@tue.nl
[2] University of Paderborn, Paderborn, Germany
{vroubine,kindler}@uni-paderborn.de

Abstract. More and more information about processes is recorded by information systems in the form of so-called "event logs". Despite the omnipresence and richness of these event logs, most software vendors have been focusing on 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 on the actual business processes. Therefore, *process mining* techniques attempt to extract non-trivial and useful information from event logs. One element 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. Many control-flow discovery techniques have been proposed in literature. Unfortunately, these techniques have *problems when discovering processes with complicated dependencies*. This paper proposes a *new two-step approach*. First, a transition system is constructed which can be modified to avoid over-fitting. Then, using the "theory of regions", the model is synthesized. The approach has been implemented in the context of ProM and uses Petrify for synthesis. This paper demonstrates that this two-step approach overcomes many of the limitations of traditional approaches.

1 Introduction

Processes, organizations, and information systems are getting more and more interconnected. Organizations are "designed" around business processes and information systems. Processes and information systems have become intertwined in many ways. Today, many of the activities in a process are either supported or monitored by information systems. Therefore, there is an increased interest in the so-called *Process-Aware Information Systems* (PAISs) as a means to bridge the perceived gap between people and software through process technology [20]. ERP (Enterprise Resource Planning), WFM (WorkFlow Management), CRM (Customer Relationship Management), SCM (Supply Chain Management), and PDM (Product Data Management) software can be used to create PAISs. Not

only WFM systems have a process engine; modern ERP, CRM, and PDM systems also have a workflow component built-in. This component allows for the design and enactment of case-driven processes, i.e., structured processes that handle cases (e.g., job applications, insurance claims, customer orders, business trips, etc.).

In recent years, it has become evident that the classical "workflow paradigm" only fits a selected subset of highly structured processes. However, one cannot assume that a PAIS supports only "production-like" processes. It can be observed that today's information systems increasingly support more dynamic processes. For example, one can use SAP without strictly following the process steps defined in the idealized reference model. Other examples are the "careflows" in hospitals where it is obvious that there is a constant need to deviate from standard processes. Therefore, even traditional workflow products started to support case handling, ad-hoc processes, and data-driven workflows. This does not imply that thinking in processes is not important. Process thinking is essential. However, *the goal is not to enforce processes but to support, monitor, and influence them.* At the same time, today's information systems record enormous amounts of data. ERP, WFM, CRM, SCM, and PDM software provide excellent logging facilities, i.e., there is a tight coupling between processes and information systems even if the processes are not enforced by the information system, i.e., information systems are *aware* of processes even if they do not control them in every aspect. Therefore, these systems are called PAISs [20]. Let us consider some examples:

- For many years hospitals have been working on Electronic Patient Records (EPR), i.e., information about the health history of a patient, including all past and present health conditions, illnesses and treatments are managed by the information system. 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 the hospital information system. 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.

– Other examples can be found in the classical administrative systems of large organizations using e.g. ERP, WFM, 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. The abundance of event data combined with the need to monitor and influence less structured processes (i.e., allowing for some form of flexibility) explains our interest in process mining.

Assuming that we are able to log events, a wide range of *process mining techniques* comes into reach [5, 6, 9, 13, 17, 18, 35]. The basic idea of process mining is to learn from observed executions of a process and (1) to *discover* new models (e.g., constructing a Petri net that is able to reproduce the observed behavior), (2) to check the *conformance* of a model by checking whether the modeled behavior matches the observed behavior, and (3) to *extend* an existing model by projecting information extracted from the logs onto some initial model (e.g., show bottlenecks in a process model by analyzing the event log). All three types of analysis have in common that they assume the existence of some *event log*.

This paper will focus on a *new type of process discovery* which entails a *two-step* approach: (1) a transition system is used as an intermediate representation and (2) a Petri net obtained through regions [7, 8, 11, 12, 21] as a final representation. Transition systems are the most basic representation of processes, e.g., simple processes tend to have many states (cf. "state explosion" problem in verification). However, using the "theory of regions" and tools like Petrify [12], transition systems can be "folded" into more compact representations, e.g., Petri nets [14, 30]. 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.

Our two-step approach overcomes many of the limitations of existing algorithms. 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.

– *Over-generalize.* Many algorithms have a tendency to over-generalize, i.e., the discovered model allows for much more behavior than actually recorded in the log. In some circumstances this may be desirable. However, there seems to be a need to flexibly balance between "overfitting" and "underfitting".
– *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]).

We will show that the approach presented in this paper is able to address these problems.

This paper will not address issues related to 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"). Heuristics [35] or genetic algorithms [2] can deal with noise, but are outside the scope of this paper. Nevertheless, it should be easy to combine the ideas in this paper with existing approaches already dealing with noise.

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 [19]. For the second step of our approach ProM calls Petrify [12] to synthesize the Petri net. Petrify implements different algorithms for computing a Petri net from a transition system. Moreover, it can be configured to calculate Petri nets with specific syntactic constraints.

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. 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 and too generic.

2.1 Overview of Process Mining

Process mining is applicable to a wide range of information systems. There are different kinds of Process-Aware Information Systems (PAISs) [20] that produce *event logs*. Examples are classical workflow management systems (e.g. Staffware), ERP systems (e.g. SAP), case handling systems (e.g. FLOWer), PDM

systems (e.g. Windchill), CRM systems (e.g. Microsoft Dynamics CRM), middleware (e.g., IBM's WebSphere), hospital information systems (e.g., Chipsoft), etc. These systems provide very detailed information about the activities that have been executed. The goal of process mining is to extract information (e.g., process models) 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 "process owners" 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.

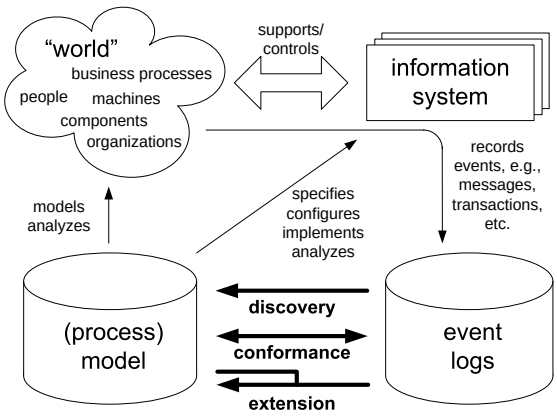

Fig. 1. Three types of process mining: (1) Discovery, (2) Conformance, and (3) Extension.

The idea of process mining is to discover, monitor and improve real processes (i.e., not assumed processes) by extracting knowledge from event logs. Clearly process mining is relevant in a setting where much flexibility is allowed or needed and therefore this is an important topic in this paper. The more ways in which people and organizations can deviate, the more variability and the more inter-

esting it is to observe and analyze processes as they are executed. 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 [5] a process model can be discovered based on low-level events. events.
- **Conformance**: There is an a-priori model. This model is used to check if reality conforms to the model. 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.
- **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 to project the bottlenecks on.

Traditionally, process mining has been focusing on *discovery*, i.e., deriving information about the original process model, the organizational context, and execution properties from enactment logs. An example of a technique addressing the control flow perspective is the α-algorithm, which constructs a Petri net model [14, 16, 30] describing the behavior observed in the event log. However, process mining is not limited to process models (i.e., control flow) and recent process mining techniques are more and more focusing on other perspectives, e.g., the organizational perspective or the data perspective. For example, there are approaches to extract social networks from event logs and analyze them using social network analysis [3]. This allows organizations to monitor how people and groups are working together.

Conformance checking compares an a-priori model with the observed behavior as recorded in the log. In [31] it is shown how a process model (e.g., a Petri net) can be evaluated in the context of a log using metrics such as "fitness" (Is the observed behavior possible according to the model?) and "appropriateness" (Is the model "typical" for the observed behavior?). However, it is also possible to check conformance based on organizational models, predefined business rules, temporal formula's, Quality of Service (QoS) definitions, etc.

There are different ways to *extend* a given process model with additional perspectives based on event logs, e.g., decision mining [32]. Decision mining, also referred to as decision point analysis, aims at the detection of data dependencies that affect the routing of a case. Starting from a process model, one can analyze how data attributes influence the choices made in the process based on past process executions. Classical data mining techniques such as decision trees can be leveraged for this purpose. Similarly, the process model can be extended with timing information (e.g., bottleneck analysis).

At this point in time there are mature tools such as the ProM framework [19], featuring an extensive set of analysis techniques which can be applied to real process enactments while covering the whole spectrum depicted in Figure 1.

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. Typically, we assume event logs that contain information about the execution of activities for cases (i.e., process instances). Figure 2 shows a small fragment of a log in MXML, the format used by ProM [19].

Fig. 2. A small fragment of an event log in MXML format showing information about processes, cases (ProcessInstance field), activities (WorkflowModelElement field), resources (Originator field), transactional information, timing information, and data elements.

The ProM Import Framework 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 [23]. 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 Web-

Sphere, BI tools like ARIS PPM, etc. Moreover, it is been used to develop many organization/system-specific conversions (e.g., hospitals, banks, governments, etc.).

Figure 2 shows the typical data present in most event logs. Note that it is not required that systems need to log all of this information, e.g., some systems do not record transactional information (e.g., just the completion of activities is recorded), related data, or timestamps. In the MXML format used by ProM, everything is optional except the ProcessInstance field and the WorkflowModelElement field, i.e., *any event needs to be linked to a case (process instance) and an activity*. All fields are relevant, however, since we focus on control-flow discovery, we can restrict ourselves to these two mandatory fields and assume that an event is described by a pair (c, a) where c refers to the case and a refers to the activity. In fact, we can further simplify things by not considering dependencies between cases. We assume that each of the cases is executed independent 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 simply focus on the ordering of activities within individual cases*. Therefore, a single case σ can be described by a sequence of activities, i.e., a trace $\sigma \in A^*$ where A is the set of activities. Consequently, a log can be described by a set of traces.

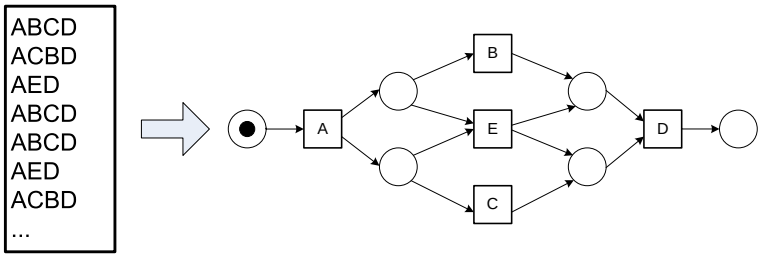

Fig. 3. A log represented by sequences of activities and the process model that is discovered using the α-algorithm.

Figure 3 shows an example of a log and the corresponding process model that can be discovered using techniques such as the α-algorithm [5]. 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.[3] Note that the α-algorithm is a very simple algorithm, but like many other algorithms it has several limitations. In Section 6 we mention some of these algorithms.

[3] In this paper, we assume that the reader has a basic understanding of Petri nets, cf. [14, 16, 30].

As indicated in the introduction, existing process mining algorithms for control-flow discovery typically have several problems. Using the example shown in Figure 3, 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 3 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 [14]. However, in reality processes tend to be non-free-choice. In this particular example, the α-algorithm [5] is able to deal with the non-free-choice construct. However, some algorithms are unable to deal with this example and 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*. 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. Consider for example Figure 3 and assume that activities A and D are both recorded as X. For example, the trace $ABCD$ 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. However, if A and D really play a different role in the process, algorithms that are unable to separate them will run into all kinds of problems, e.g., the model becomes more difficult or incorrect.

The third problem is that many algorithms have a tendency to *over-generalize*, i.e., the discovered model allows for much more behavior than actually recorded in the log. We will discuss this in more detail in Section 2.3 when we discuss the completeness of a log.

The fourth 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 [5] 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]. 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, 5] for details.

The four problems just mentioned illustrate the need for more powerful algorithms. See also [28] for a more elaborate discussion on these and other challenges in control-flow discovery. This is the reason we propose a new approach in this paper.

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 3, 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 occur in the log. Therefore, process mining is always based on some notion of completeness. A mining algorithm could be very precise in the sense that it assumes that only the sequences in the log are possible. This implies that the algorithm actually does not provide more insights than what is already in the log. It seems better to use Occam's Razor, i.e., "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. Therefore the α-algorithm [5] 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.

To illustrate the relevance of completeness, consider 10 tasks which can be executed in parallel. The total number of interleavings is 10! = 3628800. It is probably not realistic that each interleaving is present in the log. However, for local completeness only 10(10-1)=90 observations are needed.

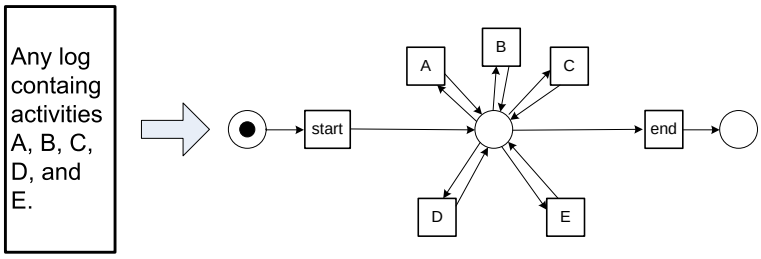

Fig. 4. The so-called "flower Petri net" allowing for any log containing A, B, C, D, and E.

Different algorithms assume different notions of completeness. These notions illustrate the different attempts 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 it 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 4. This so-called "flower Petri net" allows for any sequence starting with *start* en 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., $AADD$.

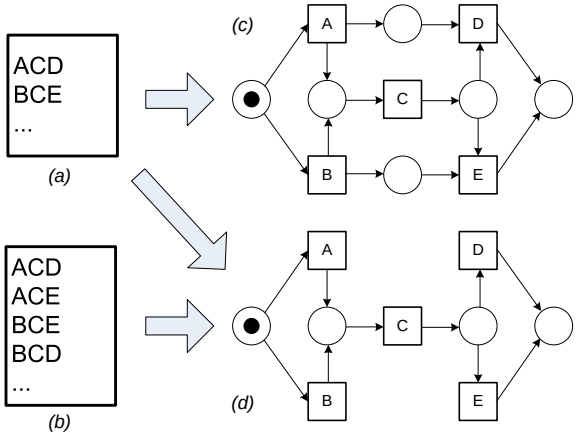

Fig. 5. Two logs and two models illustrating issues related to completeness (i.e., "overfitting" and "underfitting").

Let us now consider another example showing that it is difficult to balance between being too general and too specific. Figure 5 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 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.

Figure 5 shows that there is a delicate balance and that it is non-trivial to compare logs and process models. In [31] 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. In other words: the Petri net should be able

to "parse" (i.e., reproduce) every activity sequence observed. In [31] 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 network and not all transactions in the model need to be logged [31]. Unfortunately, a good fitness only 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 4). Although such Petri nets have a fitness of 1, they do not provide meaningful information. Therefore, in [31] a second dimension is introduced: *appropriateness*. Appropriateness tries to answer the following question: "Does the model describe the observed process in a suitable way?" and can be evaluated from both a *structural* and a *behavioral* perspective. In [31] 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 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.

Despite the fact that there are different ways to quantify notions such as fitness and appropriateness, there is typically not one optimal model. *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 allowing for different interpretations of completeness to avoid "overfitting" and "underfitting".

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 this transition system is synthesized into a Petri net. This section describes the first step. An important quality of the first step is that, unlike existing approaches, it can be tuned towards the application. Depending on the desired qualities of the model, the algorithm can be tuned to provide suitable models.

3.1 Preliminaries

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

Let A be a set. $\mathbb{B}(A) = A \to \mathbb{N}$ is the set of multi-sets (bags) over 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 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: $set(X) = \{a \in X \mid X(a) > 0\}$. Note that we assume the multi-sets to be finite.

$\mathcal{P}(A)$ is the powerset of A, i.e., $X \in \mathcal{P}(A)$ if and only if $X \subseteq A$.

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 $1 \leq i \leq n$. $hd(\sigma, k) = \langle a_1, a_2, \ldots, a_k \rangle$, i.e., the sequence of just the first k elements (where $0 \leq k \leq n$). Note that $hd(\sigma, 0)$ is the empty sequence. $tl(\sigma, k) = \langle a_{k+1}, a_{k+2}, \ldots, a_n \rangle$, i.e., the sequence after removing the first k elements $0 \leq k \leq n$. $tl(\sigma, 0) = \sigma$ and $tl(\sigma, n)$ is the empty sequence. $\sigma \uparrow X$ 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 $\langle d, a, a, a, a, a, a, d \rangle \uparrow \{d\} = \langle d, d \rangle$.

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 $a \in A$: $par(\sigma)(a) = |\sigma \uparrow \{a\}|$.

The Parikh vector will be used to count the number of times activities occur in a sequence of activities.

3.2 Approach

In Section 2.2 we discussed the MXML format used in our tools (cf. Figure 2). Although MXML can store transactional information, information about resources, related data, and timestamps, we 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 by a sequence of activities and a log can be described by a set of traces[4].

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

Note that $a \in A$ may refer to an atomic activity or may be structured, e.g., the set of documents produced in an activity.

The set of activities can be found by inspecting the log. The most important aspect of *process mining is however deducing the states of the process*. Most mining algorithms have an implicit notion of state, i.e., activities are glued together in some process modeling language based on an analysis of the log and the resulting model has a behavior that can be represented as a transition system. In

[4] 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 is suffices to consider sets.

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 state information from the activities executed before and after a given state. Based on this we conclude that, when building a transition system, there are basically four approaches to determine the state 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,
- *past and future*, i.e., a combination of the previous two, or
- *explicit knowledge* of the current state, e.g., the log contains state information in addition to event data.

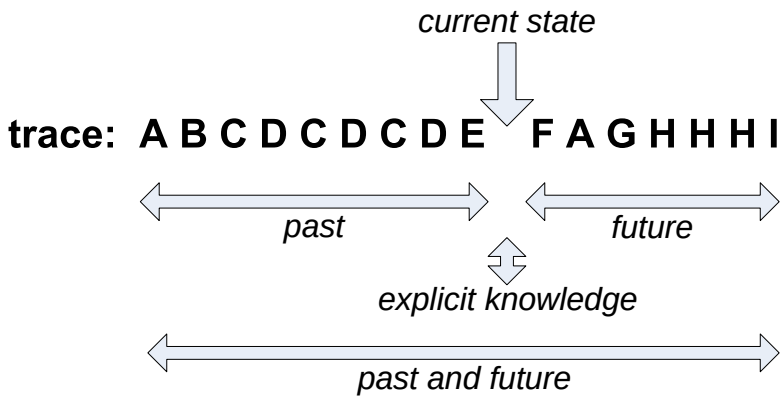

Fig. 6. At least four basic "ingredients" can be considered as a basis for calculating the "process state": (1) past, (2) future, (3) past and future, and (4) explicit knowledge.

Figure 6 shows an *example* of a trace and the different "ingredients" that can be used to calculate state information.

In this paper, we assume that we do not have *explicit knowledge* about the current state and focus on the *past* and *future* of a case. However, note that our approach can also be applied to situations where we have explicit state knowledge [26].

Definition 2 (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 $hd(\sigma, k)$. The* future *of this case after executing k steps ($0 \leq k \leq n$) is $tl(\sigma, k)$.*

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 we will see in the remainder.

First of all, the basis of calculation may be a complete or partial prefix (postfix):

- *complete prefix (postfix)*, i.e., the state is represented by a complete history (future) of the case,
- *partial prefix (postfix)*, i.e., only a subset of the trace is considered.

A partial prefix only looks at a limited number of events before the state is reached. For example, while constructing the state information for the purpose of process mining, one can decide to only consider the last k events. For example, instead of taking the complete prefix $\langle A, B, C, D, C, D, C, D, E\rangle$ shown in Figure 6 only the last four ($k = 4$) events are considered: $\langle D, C, D, E\rangle$. In a partial postfix also a limited horizon is considered, i.e., seen from the state under consideration only the next k events are taken into account.

Second, only a selected set of activities may be considered, i.e., the log is filtered and only the remaining events are used as input for process mining. This is another type of abstraction orthogonal to taking a partial prefix (postfix). Filtering may be used to remove certain activities. For example, if there are start and complete events for activities, e.g., "A started" and "A completed", then it is possible to only consider the complete events. It is also possible to filter out infrequent activities and focus on the frequent activities to simplify the discovered model. Filtering is a very important abstraction mechanism in process mining. Therefore, tools such as ProM provide a wide range of filtering tools taking transaction types, data fields, resources, and frequencies into account.

The third abstraction mechanism 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 occurred at some time in the past 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/or 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.

Figure 7 illustrates the different ways of representing the knowledge about the past or future for the purpose of process mining. Note that the different kinds of abstraction can be combined (assuming that we do not have explicit knowledge). This results in 3 (past/future/past and future) times 2 (complete/partial) times 2 (no filter/filter) times 3 (sequence/multi-set/set) $= 3 * 2 * 2 * 3 = 36$ strategies

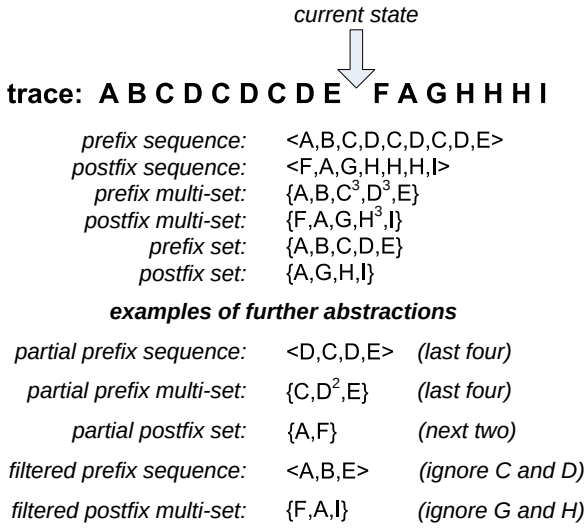

Fig. 7. There are many ways to construct the "current state" depending on the desired level of abstraction.

to represent states. 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.

Let us now try to further operationalize the ideas illustrated by Figure 7. Definition 2 already showed how to project a sequence of activities onto the past or future using $hd(\sigma, k)$ and $tl(\sigma, k)$ (given a trace σ and the state resulting after k steps). The operators *par* and *set* defined in Section 3.1 can be used to abstract away the ordering of activities thus map sequences onto sets or multi-sets. To filter, we use $\uparrow$ defined in Section 3.1.

Definition 3 (Filtering). *Let A be a set of activities, $X \subseteq A$ a subset of activities, and $\sigma \in A^*$ a trace. $\sigma \uparrow X$ is the sequence obtained by removing the events that are not in X, i.e., σ is filtered by only keeping the activities in X.*

When considering partial pre/postfixes, we need to define a horizon h and use $hd^h(\sigma, k)$ and $tl^h(\sigma, k)$ rather than $hd(\sigma, k)$ and $tl(\sigma, k)$ as defined below.

Definition 4 (Horizon). *Let A be a set of activities and $\sigma = \langle a_1, a_2, \ldots, a_n \rangle \in A^*$ a complete trace of length n. Let h be a natural number defining the horizon and let k ($0 \leq k \leq n$) point to the current state in the trace σ (i.e., state state after executing k steps). The partial prefix $hd^h(\sigma, k) = \langle a_{(k-h)\ max\ 1}, \ldots, a_k \rangle$ is the sequence of at most h events before reaching the current state. The partial postfix $tl^h(\sigma, k) = \langle a_{k+1}, \ldots, a_{(k+h)\ min\ n} \rangle$ is the sequence of at most h events following directly after the current state.*

As indicated before, the representation of the current state can be very detailed or not. For example, given a trace σ after k steps, the state may be represented as $(hd(\sigma, k), tl(\sigma, k))$, i.e., the current state is represented by the complete prefix and postfix sequences. However, to avoid "overfitting" other representations can be used. The representation $par(hd(\sigma, k))$ only considers the complete prefix multi-set, i.e., the full prefix is considered but the ordering is not relevant. The representation $set(par(hd(\sigma, k)))$ considers the complete prefix set, i.e., the full prefix is considered, the ordering is not relevant, and the frequency is not relevant. Another example state representation is $set(par(tl^h(\sigma, k)))$ which considers a partial postfix of length h without caring about ordering a frequencies. $set(par(tl^h((\sigma \uparrow X), k)))$ is similar but now first the sequence is filtered and all activities not in X are removed. After these examples, we define the concept of state representation with respect to a position in trace explicitly.

Definition 5 (State representation). *A state representation* state *is a function which, given a sequence σ and a k indicating the number of events of σ that have occurred, produces a some representation r. Formally, state $\in (A^* \times \mathbb{N}) \not\to R$ where A is the set of activities, R is the set of possible representations (e.g., sequences, sets, or bags over A), and $dom(state) = \{(\sigma, k) \in A^* \times \mathbb{N} \mid 0 \leq k \leq |\sigma|\}$.*

An example of a state representation is: $state(\sigma, k) = set(par(tl^h((\sigma \uparrow X), k)))$.

The various abstraction concepts can be used to "tune" the state representation. Many different *state* functions are possible and here we only list the obvious ones. As was indicated before, we consider $3*2*2*3 = 36$ strategies to represent states. Each of these strategies defines a *state* function. These can be constructed as follows. Assume a complete trace σ and a k indicating the current position in σ.

1. Filter the log if needed, i.e., use σ or $\sigma \uparrow X$ as a basis. (Two possibilities.)
2. Decide to use just the past, just the future, or both and determine if partial or complete pre/postfixes are used. There are six possibilities: $hd(\sigma, k)$, $tl(\sigma, k)$, $(hd(\sigma, k), tl(\sigma, k))$, $hd^h(\sigma, k)$, $tl^h(\sigma, k)$, and $(hd^h(\sigma, k), tl^h(\sigma, k))$.
3. Determine if the ordering and frequency of activities is relevant and further abstract from this in the resulting post/prefixes if needed. Assuming a pre/postfix σ it is possible to retain the sequence σ, to remove the ordering $par(\sigma)$ (i.e., construct a multi-set), or to remove also the frequencies $set(par(\sigma))$ (i.e., construct a set). (Three possibilities.)

One of the 36 possible strategies is for example:

$$state(\sigma, k) = (set(par(hd^h((\sigma \uparrow X), k))), set(par(tl^h((\sigma \uparrow X), k))))$$

The next step is to build a transition system based on a particular *state* function. The state space is given by all the states visited in the log when assuming the representation chosen. The transition relation can be derived by assuming that one can go from one state to another if this occurs in at least one of the traces in the log.

3.3 Constructing a Transition System (Step 1a)

In this section, we give a general definition of a *transition system*, which is based on the notion of *state* presented in Definition 5.

Definition 6 (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, 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, $E = A$ is the set of events (labels) and $T \subseteq S \times E \times S$ with $T = \{(state(\sigma, k), \sigma(k+1), state(\sigma, k+1)) \mid \sigma \in L \ \wedge \ 0 \leq k < |\sigma|\}$ is the transition relation.*

This definition shows that we can build different transition systems depending on the type of state representation selected. Note that we identified more than 36 types in Section 3.2. 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 $state(\sigma, k-1) \xrightarrow{\sigma(k)} state(\sigma, k)$ and these are added if it does not exist yet[5]. It is worth mentioning that when dealing with the states based on the complete prefix, a transition system can be constructed effectively online just while reading a log.

As an example, let us take the following log:

$$\mathbf{L} = \left\{ \begin{array}{l} \langle A, B, C, D \rangle, \\ \langle A, C, B, D \rangle, \\ \langle A, E, C, E, C, D \rangle \end{array} \right\} \tag{1}$$

If we use the *complete prefix set* definition of a state, i.e. $state(\sigma, k) = set(par(hd(\sigma, k)))$, we get the transition system shown in Figure 8. Every state consists of a set of activities and every transition is labeled with a name of an activity. This transition system contains two self-loop transitions $\{A, C, E\} \xrightarrow{E} \{A, C, E\}$ and $\{A, C, E\} \xrightarrow{C} \{A, C, E\}$. If we use the *complete prefix sequence* representation of a state, i.e. $state(\sigma, k) = hd(\sigma, k)$, we obtain another transition system as shown in Figure 9. For this transition system, every state is represented by a sequence of activities (for example $\langle A, E, C, E \rangle$). As is easy to see, this transition system does not contain any self-loops anymore. In fact, the complete prefix sequence representation of a state always results in acyclic transition systems. Another example of a transition system generated based on the example log is shown in Figure 10. This one is based on the *complete postfix multi-set* definition of a state.

3.4 Modifications (Step 1b)

Transition systems, which are constructed according to the algorithm given in the previous section, reflect the behavior seen in the log. However, often the log

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

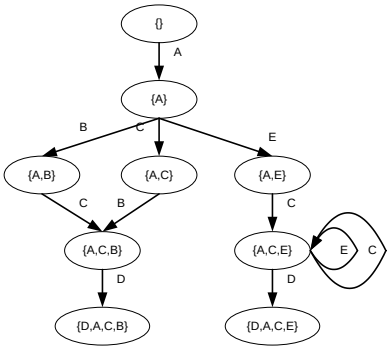

Fig. 8. A transition system constructed using complete prefix sets.

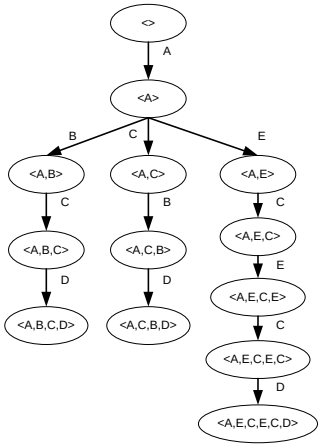

Fig. 9. A transition system constructed using complete prefix sequences.

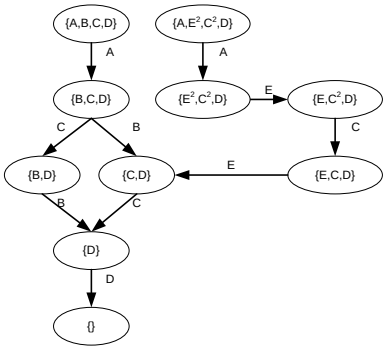

Fig. 10. A transition system constructed using complete postfix multi-sets.

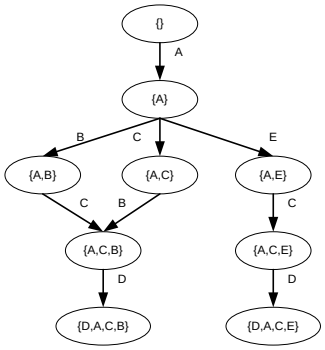

Fig. 11. A transition system that is made acyclic using the "Kill Loops" strategy.

does not contain all the possible traces and, thus, represents only a part of the possible behavior. In the other cases, we need to further *abstract* from the log data, introduce *generalizations*, or even *ignore* some unnecessary details. So, in this section, we present the main operations which make up a framework for building "clever" modification strategies and show some examples of them.

Let $TS = (S, E, T)$ be a transition system constructed for some log $L \in \mathcal{P}(A^*)$ using a particular *state* function. The *main operations* for building strategies are:

addArc Operation $addArc(s_1, a, s_2)$ adds a new transition to the transition relation T, i.e., an arc labeled a connecting state s_1 to state s_2. $TS' = (S, E, T')$ with $T' = T \cup \{(s_1, a, s_2)\}$ is the transition system with an added arc (assuming $(s_1, a, s_2) \notin T$).

removeArc Operation $removeArc(s_1, a, s_2)$ removes a transition. $TS' = (S, E, T')$ with $T' = T \setminus \{(s_1, a, s_2)\}$ is the transition system without this arc (assuming $(s_1, a, s_2) \in T$).

mergeStates Operation $mergeStates(s_1, s_2)$ creates a new state $s_{12} = s_1 + s_2$. For any state s, $T_s^I = \{(s', a, s'') \in T \mid s'' = s\}$ is the set of incoming transitions, $T_s^O = \{(s', a, s'') \in T \mid s' = s\}$ is the set of outgoing transitions, and $T_s = T_s^I \cup T_s^O$ is the set of all incident transitions. The transition system resulting from operation $mergeStates(s_1, s_2)$ is $TS' = (S', E, T')$ with $S' = (S \setminus \{s_1, s_2\}) \cup \{s_{12}\}$, $T' = (T \setminus (T_{s_1} \cup T_{s_2})) \cup T_{new})$, where $T_{new} = \{(s, a, s_{12}) \mid \exists_{s'} (s, a, s') \in T_{s_1}^I \cup T_{s_2}^I\} \cup \{(s_{12}, a, s) \mid \exists_{s'} (s', a, s) \in T_{s_1}^O \cup T_{s_2}^O\}$.

Next, we present some useful strategies, and show how they can be applied to our examples. Note that these strategies can be used in combination with the many strategies defined for the state representation. Moreover, these are just examples showing that the $addArc(s_1, a, s_2)$, $removeArc(s_1, a, s_2)$, and $mergeStates(s_1, s_2)$ operations can be used to "massage" the transition system before constructing a process model from it.

"Kill Loops" Strategy The "Kill Loops" strategy is used for ignoring the loops and, thus, for building *acyclic* transition systems. When a set representation is used, typically self-loops are introduced (whenever an activity is executed for the second time a self-loop is created). See for example the TS shown in Figure 8 that has two self-loop transitions $\{A, C, E\} \xrightarrow{E} \{A, C, E\}$ and $\{A, C, E\} \xrightarrow{C} \{A, C, E\}$. Let $TS = (S, E, T)$ be a transition system where self-loops need to be removed. $TS' = (S, E, T')$ with $T' = \{(s_1, a, s_2) \in T \mid s_1 \neq s_2\}$ is the resulting transition system. A transition system derived after applying this strategy to the set-based transition system given in Figure 8 is shown in Figure 11.

The "Kill Loops" strategy is motivated that in some cases one is interested only in the occurrence of an activity and not the ordering of activities or the frequency of an activity. Hence, sets are used to represent states. However, a side-effect of the set representation is the introduction of self-loops for activities that can occur multiple times. These can effectively be removed using this strategy (but this changes the initial behaviour).

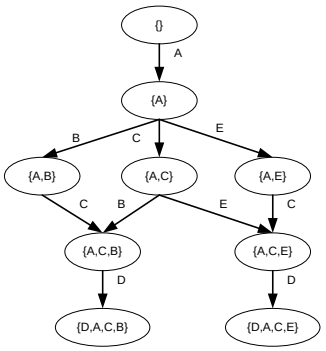

Fig. 12. Result of applying the extend strategy.

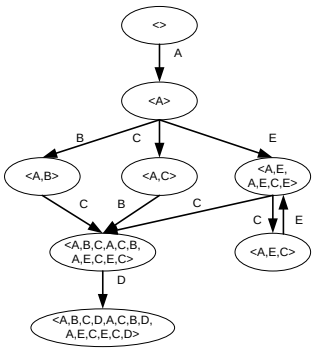

Fig. 13. Result of applying the merge states strategy.

"Extend" Strategy The "Extend" strategy is especially useful for the logs having a set representation. Let $TS = (S, E, T)$ be a transition system where this strategy has to be applied. $TS' = (S, E, T')$ with $T' = T \cup \{(s_1, a, s_2) \in S \times E \times S \mid s_1 \cup \{a\} = s_2 \wedge a \notin s_1\}$ is the resulting transition system. Basically, this strategy makes transitions between two states, which were created from different traces but which can be subsequent because there is a single activity which can be executed to reach one state from the other. This strategy is very useful for generalizing the behavior seen in the logs by means of extending the "state diamonds", i.e., interleavings are added to allow for the deduction of parallel constructs. An example of this strategy applied to the acyclic transition system of Figure 11 is shown in Figure 12. A transition $\{A, C\} \xrightarrow{E} \{A, C, E\}$ was added here. It should be noted that a combination of strategies is used, i.e., both "Kill Loops" and "Extend" are applied to the original transition system shown in Figure 8.

The motivation for the "Extend" strategy is that, in many cases, it is unrealistic that all possible interleavings of activities are actually present in the log. When discussing the notion of completeness, we demonstrated that this is indeed a problem (Section 2.3). Therefore, the "Extend" strategy in a way extends the transition system with interleavings not observed in the log but likely to be present based on the structure. The "Extend" strategy is often used in combination with a set representation of states. This representation seems natural when in each activity some artifact, e.g. a document, is produced. In this case, the state refers to the set of artifacts produced so far, e.g., all documents that have been checked in. In this context the assumption of the "Extend" strategy is that it is possible to move from one state to anther if there is a difference of a single document, i.e., if $\{a_1, \ldots, a_n\}$ and $\{a_1, \ldots, a_n, a_{n+1}\}$ are reachable states, then there is a transition from $\{a_1, \ldots, a_n\}$ to $\{a_1, \ldots, a_n, a_{n+1}\}$.

"Merge by Output" Strategy Another useful strategy is called "Merge by Output". It merges the states that have the same outputs. This strategy is especially useful for dealing with loop constructs. Let $TS = (S, E, T)$ be a transition system. For any state s let us define an operation $out(s) = \{a \in E \mid (s, a, s'') \in T\}$, which returns the set of output events of a state. Let us define a predicate $isMerge \subseteq S \times S$ such that for any $s_1, s_2 \in S$: $isMerge(s_1, s_2)$ if and only if $out(s_1) = out(s_2)$. If $isMerge(s_1, s_2)$, then s_1 and s_2 are merged onto a new state. $M_{TS} = \{(s_1, s_2) \mid isMerge(s_1, s_2)\}$ contains all the pairs of states that can be merged. For any pair of states $(s_1, s_2) \in M_{TS}$, we can execute a $mergeStates(s_1, s_2)$ operation, which produces a new transition system TS', according to the definition given above. Based on this new transition system TS', we can again calculate all the pairs of states that can be merged $M_{TS'}$. Again a pair of states is selected and merged using the $mergeStates$ operation. This is repeated until there are no more states to be merged.

There are several ways to refine the $isMerge$ predicate. For example, function out could be refined to not only take into account the output event but also the output state. Another refinement would be to avoid merging states if this introduces loops, e.g., we can redefine the predicate $isMerge$ to: $isMerge(s_1, s_2)$ if and only if $out(s_1) = out(s_2)$ and $\nexists\, a, b \in E :\ (s_1, a, s_2) \in T$ or $(s_2, a, s_1) \in T$, or there exists an s'' such that $(s'', a, s_1), (s'', b, s_2) \in T$. The additional requirements are given to prohibit building self-loops and multiple arcs between a pair of states in a transition system after merging.

If we take the sequence-based transition system shown in Figure 9 and assume the more refined $isMerge$ predicate, the set M_{TS} includes pairs such as ($\langle A, E\rangle$, $\langle A, E, C, E\rangle$), ($\langle A, B, C\rangle, \langle A, E, C, E, C\rangle$), ($\langle A, B\rangle$, $\langle A, E, C, E\rangle$), and ($\langle A, B, C, D\rangle$, $\langle A, E, C, E, C, D\rangle$). Note that after merging a pair of states, the set $M_{TS'}$ of the new transition system TS' will be different from M_{TS}. So, starting with merging the pair $(\langle A, E\rangle, \langle A, E, C, E\rangle)$, and then producing new transition system and merging the states there, finally (when no states can be merged) we produce a transition system shown in Figure 13.

In our example, we use only the "merge by output" strategy; but in general, the strategies based on equality of inputs and subsets of outputs or inputs are very helpful for simplifying the transition system and solving the problem of "loops". In this section, we defined the basic framework for building strategies and presented the motivating examples of their potential for simplifying the transition systems. However, in future, further ideas about strategies, their combinations and their applications should be worked out.

Clearly, the three strategies presented in this section ("Kill Loops", "Extend", and "Merge by Output") are just examples. Moreover, even for these three strategies many variants exist. It is also important to note that the suitability of a strategy heavily depends on the state representation selected. There are numerous combinations possible, some of which work better than others depending on the characteristics of the event logs at hand. This differentiates our approach for existing approaches which typically propose a single algorithm that cannot be configured to address different needs.

4 Synthesis using Regions (Step 2)

In this section, we present the second step of our approach. In this second step a Petri net is synthesized from the transition system resulting from the first step. To do this we use the "theory of regions" [21, 15, 12].

4.1 Constructing Petri Nets Using Regions

First, we give the classical definition of a *region*.

Definition 7 (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 $s_1, s_2 \notin S'$*

In Figure 14, we continue the example of the sets-based transition system with killed loops (see Figure 11) and present several examples of regions. The set $r_0 = \{\{\}\}$ is a region, since all the transitions labeled with A exit it and all other labels do not cross. It is important to see that r_0 is a set of states containing one state being the empty set. $r_1 = \{\{A\}, \{A, C\}\}$ is a region, since A enters it, B and E exit it and C and D do not cross it; $r_2 = \{\{A, B\}, \{A, E\}, \{A, C, B\}, \{A, C, E\}\}$ and $r_3 = \{\{D, A, C, B\}, \{D, A, C, E\}\}$ are two other examples of regions, they are also marked with dotted lines in the figure.

Any transition system $TS = (S, E, T)$ has two trivial regions: $\emptyset$ (the empty region) and S (the region consisting of all states). In the remainder we only consider non-trivial regions.

A region r' is said to be a *subregion* of the other region r if $r' \subset r$. For example, r_0 and r_1 are subregions of region $r = \{\{\}, \{A\}, \{A, C\}\}$. A region r is *minimal* if there is no other region r' which is a subregion of r. For example, both r_0 and r_1 are minimal regions; in Figure 14, the set of regions marked with dotted lines is the set of all the minimal regions of the TS. A region r is a *preregion* of event e if there is a transition labeled with e which exits r. A region r is a *postregion* of event e if there is a transition labeled with e which enters r. For example, r_0 is a preregion of A and r_1 is a postregion of A.

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 our transition system is given in Figure 15. The incoming place of the transition A corresponds to the minimal region r_0 and the

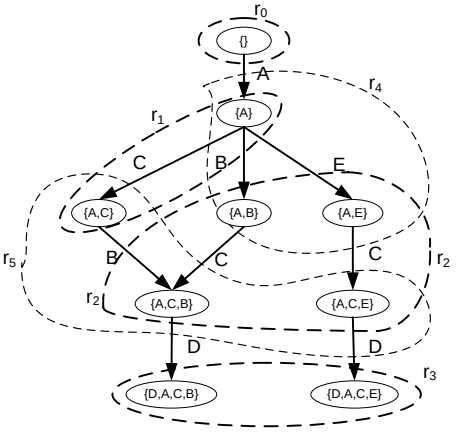

Fig. 14. Regions in the transition system.

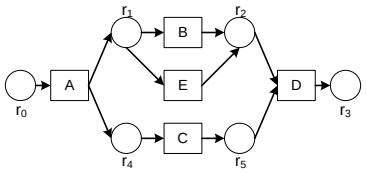

Fig. 15. Synthesized PN (algorithm for elementary TS).

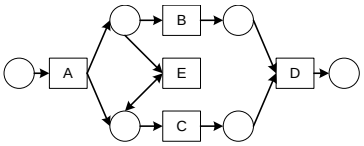

Fig. 16. Synthesized PN (algorithm for non-elementary TS).

outgoing place of A which is also the incoming place for transitions B and E corresponds to the region r_1 respectively.

The first papers on the theory of regions only dealt with a special class of transition systems called *elementary transition systems*. See [15, 7, 8] for details. The class of elementary transition systems is very restricted. In practice, most of the time, people deal with arbitrary transition systems that only by coincidence fall into the class of elementary transition systems. In the papers of Cortadella et al. [11, 12], a method for handling any transition system 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.

In our example, the TS shown in Fig. 14 is not an elementary transition system, whereas the synthesis algorithm described before produces appropriate results for the elementary ones. Thus, the PN in Fig. 15 allows for more behavior than we have seen in the TS. The result of applying the algorithm of Cortadella et al. is shown in Fig. 16, it exactly corresponds to the TS from Fig. 14 .

Most of the transition systems shown in the examples from Section 3 are not elementary. However, from a practical point of view this is just a technicality that can easily be resolved. In the remainder of this paper, we build our approach on the approach of Cortadella et al. [12]. We start our examples with the Petri nets synthesized from the basic transition systems, which were constructed in Section 3.3. The Petri net shown in Figure 17 was synthesized from the transition system shown in Figure 8 and the Petri net in Figure 18 from Figure 9 correspondingly.[6] Both Petri nets reflect the behaviour seen in the log (logs can be successfully replayed in this Petri nets), but they also have some disadvantages: the first Petri net is too general, because transitions C and E can be executed an unlimited number of times; the second Petri net is too explicit, since for the last trace (see the log used for the running example) it allows only the sequence $\langle A, E\_1, C, E, C, D\rangle$ that is presented in the log, but not the loop construct.

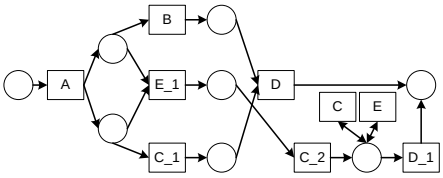

Fig. 17. Petri net for the transition system based on sets.

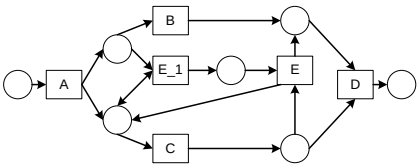

Fig. 18. Petri net for transition system based on sequences.

The next set of examples shows some modified transition systems constructed using the various strategies described earlier. The Petri net synthesized from the acyclic set-based transition system from Figure 11 (obtained after applying the "kill loops" strategy) is shown in Figure 16. It is compact and exactly reflects the behaviour from the log, but ignores the loop. The Petri net that corresponds to the extended transition system is shown in Figure 19, it supports additional behaviour, for example it also allows for the trace $\langle A, C, E, D\rangle$. The last Petri net is shown in Figure 20. This Petri net is derived from the sequence-based transition system, where the "merge by output" strategy was applied. This Petri net specifies the behaviour seen in the log, but also recognizes the loop.

4.2 Selecting the Target Format

In this section, we deal with different target formats of synthesized Petri nets. We use various synthesis algorithms to derive Petri nets in different ways and

[6] The Petri nets shown are labeled, so transitions denoted like $E, E\_1, E\_2, \ldots$ all refer to the same event E.

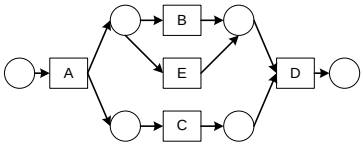

Fig. 19. Petri net for the extended transition system.

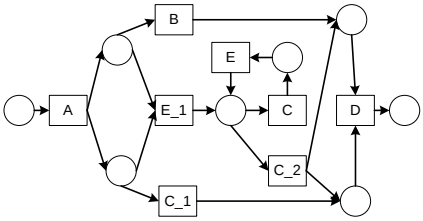

Fig. 20. Petri net for the transition system after state merging.

to produce different classes of Petri nets, such as free choice, extended free-choice, pure, state-machine decomposable and others. The algorithms synthesize *labeled Petri nets* and they are based on *transition label splitting*. This way, the Petri nets, which can be huge and difficult to understand, can be converted and simplified. Here, we use the algorithms developed in the work of Cortadella et al. [11, 12], as they generally deal with labeled Petri nets.

As described in Section 4.1, the algorithms of Cortadella et al. deal not only with elementary but also with the full class of transitions systems. So, all the algorithms check whether a transition system is elementary and split appropriate labels in case it is not elementary; the splitting is based on the notions of *excitation* and *generalized excitation region*, see [11]. The simplest synthesis algorithm generates a Petri net from all the regions, this net is called a *saturated net*. An improvement of this algorithm is generating a *minimal saturated net*, which is based on all the minimal regions. However, both algorithms produce nets with *redundant places*, i.e. some places can be removed without changing the behavior. Building a *place-irredundant net* with minimal regions is a challenging task, which can be solved by assigning costs to different solutions based on minimal regions and finding the optimal one. So, the algorithm, which was used for all the examples presented above generates a subset of all the minimal regions of a transition system, which is sufficient for Petri net synthesis. All the obtained Petri nets are *place-irredundant*.

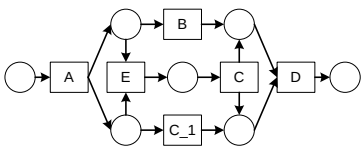

Fig. 21. Pure Petri net.

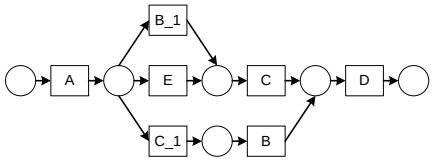

Fig. 22. Free-choice Petri net.

As mentioned above, the algorithms support synthesis of different classes of Petri nets and it is possible to control the type of the constructed net. For example, we can generate alternative representations for the set-based Petri net shown in Figure 16. Using the settings of the synthesis algorithm, we can construct the *pure*[7] Petri net shown in Figure 21. To make the net pure, transition C had to be split to exclude "self-loops". Next, we can build a *free-choice*[8] equivalent for it, see Figure 22. Transition B had to be split to exclude the choice between B and E, which coincided with the synchronization for E.

In this section, we presented the second step of our approach. We demonstrated a method for deriving Petri nets from transition systems constructed from the event logs. We have shown the benefits of using the well developed theory of regions and Petri net synthesis in the area of process mining.

5 Implementation and Evaluation

The ideas presented in this paper have been implemented in the context of *ProM*. ProM serves as a testbed for our process mining research [19] 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. Information as shown earlier in tabular form can be stored in MXML. One MXML file can store information about multiple processes. For each process, events related to particular process instances (often called cases) are stored. Each event refers to an activity. In the context of this paper documents are mapped onto activities. Events can also have additional information such as the transaction type (start, complete, etc.), the originator (the resource, e.g., person, that executed the activity, in this paper often referred to as the "author"), timestamps (when did the event occur), and arbitrary data (attribute-value pairs).

5.1 ProMimport

The ProMimport Framework [23] 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 is been used to develop many organization/system-specific conversions (e.g., hospitals, banks, governments, etc.). As will be shown later, the ProMimport Framework can also be used to extract event logs from systems such as Subversion and CVS (Concurrent Versions System) [23].

[7] A Petri net $PN = (P, T, F)$ is called pure if $(p, t) \in F$ implies that $(t, p) \notin F$, i.e. the Petri net has no self-loops.

[8] A Petri net $PN = (P, T, F)$ is called free-choice if $\forall p \in P : \ |p^\bullet| \leq 1$ or ${}^\bullet(p^\bullet) = \{p\}$, i.e. the Petri net does mixtures of choice and synchronization.

5.2 ProM and Petrify

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 plugins in the context of this paper are the mining plugins, it is important to note that there are in total five types of plug-ins:

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

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.

In Subsection 5.3, we illustrate the application of the plug-ins developed in the context of this paper. However, since there are currently more than 140 plug-ins it is impossible to give a representative overview. One of these more than 140 plug-ins is the mining plug-in that generates the transition system that can be used to build a Petri net model. Note that, for this particular approach, ProM calls Petrify [12] to synthesize the Petri net, which is a command-line tool for the synthesis of Petri nets from transition systems. Petrify is freely available from http://www.lsi.upc.edu/petrify/ and it implements the algorithms developed by Cortadella et al. [11, 12].

5.3 Comparison

While it is impossible to evaluate the possibilities of the region-based approach against all process mining algorithms implemented in ProM, we do evaluate it against two important mining algorithms: the α-algorithm [5] and multi-phase algorithm [17, 18]. For this comparison, we use an example of a log taken from a driving school. In this driving school, learners start by applying for a license. Then, in parallel, they take a theoretical exam, as well as driving lessons for either car or motorbike. After finishing the theoretical exam and the lessons, they take a practical exam, after which they do or do not receive a license. Note that it is only possible to do a practical exam for cars if the learner had car driving lessons and vice versa for a motorbike exam. For the comparison, we used a complete process log, i.e. a process log that showed all four possible executions of this process, namely car and bike combined with getting or not getting a license.

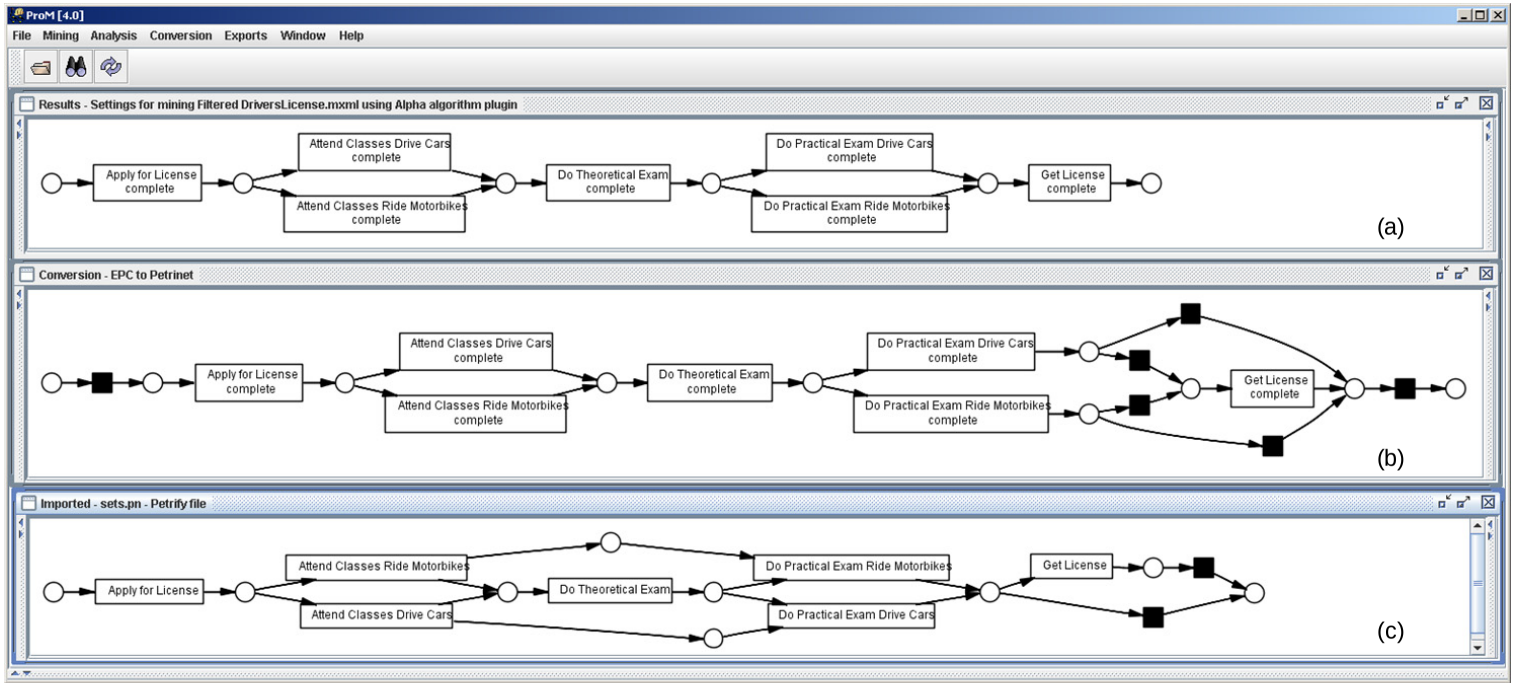

Fig. 23. ProM, showing three Petri nets, discovered using (a) the α miner, (b) the multi-phase miner and (c) the Petrify-based approach.

The first algorithm we used to generate a Petri net was the α-algorithm. The α-algorithm is a well-known process discovery algorithm, which is often used for benchmarking. The reason for this is that it is a simple algorithm that typically produces nice results if the log satisfies certain properties (cf. [5]). For our example however, the result, shown in Figure 23(a), has two problems. First of all, the resulting model allows for a learner to take car driving lessons and a motorbike exam. Second, after taking an exam, the learner always gets his license, which is even more undesirable.

Since our log does not satisfy all requirements of the α-algorithm, we use the multi-phase algorithm presented in [17, 18]. This algorithm guarantees to return a Petri net that can reproduce the log. As can be seen from Figure 23(b), it solves the problem that a learner always receives a license after the exam albeit in a rather complicated way. However, the Petri net still allows the learner to take lessons in a car and an exam on a motorbike.

This dependency between two transitions that are not directly following each other is typically hard to find by process mining algorithms. The genetic approach presented in [2, 27] is capable of finding such dependencies, but since that approach is based on genetic algorithms, it has high demands on computation time.

The result of the region-based approach presented in this paper is presented in Figure 23(c). It is clear that this Petri net indeed correctly models the process under consideration.

Although the example in this section looks rather simple, it nicely shows that the region based approach is a valuable addition to the existing collection of process discovery algorithms. However, as with any process discovery algorithm trade-offs are made with respect to the correctness of the result and the computation time. The α-algorithm and the multi-phase approach are computationally fast[9], the theory of regions approach is more complex, since it has a worst-case complexity that is exponential in the size of the log. However, the result of our approach is more accurate, since it also catches long-range dependencies, that are not detected by the multi-phase approach, nor the α-algorithm, i.e. it does not underfit. Moreover, a very important feature of the approach presented in the paper is that it is "tunable", i.e., the settings can be modified to achieve the desired result.

The results presented in this paper have been implemented in ProM and can be downloaded from www.processmining.org. In the next section, we discuss related work and provide pointers to other algorithms many of which are implemented in ProM.

[9] The multi-phase approach is polynomial in the size of the log and the α-algorithm exponential in the size of an abstraction of the log, which can be built in polynomial time.

6 Related Work

Since the mid-nineties several groups have been working on techniques for process mining [5, 6, 9, 13, 17, 18, 35], i.e., discovering process models based on observed events. In [4] 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 [6]. In parallel, Datta [13] looked at the discovery of business process models. Cook et al. investigated similar issues in the context of software engineering processes [9]. Herbst [24] 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 [5] 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 [17, 18] a more robust but less precise approach is presented.

In this paper we do not consider issues such as noise. Heuristics [35] or genetic algorithms [2, 27] have been proposed to deal with issues such as noise. It appears that some of the ideas presented in [35] can be combined with other approaches, including the one presented in this paper.

The second step in our approach uses the theory of regions [7, 8, 11, 12, 21]. 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* [10] for this purpose. The idea to use regions has been mentioned in several papers. However, only recently people have been applying regions to process mining [26]. 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., balancing between "overfitting" and "underfitting" is the most important challenge in process mining research.

Process mining can be seen in the broader context of Business Process Intelligence (BPI) and Business Activity Monitoring (BAM). In [22, 33] a BPI toolset on top of HP's Process Manager is described. The BPI toolset includes a so-called "BPI Process Mining Engine". In [29] 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) [25]. The tool is commercially available and a customized version of PPM is the Staffware Process Monitor (SPM) [34] 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.

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. The power of our approach in this paper is that it allows for a wide variety of strategies. First of all, we defined 36 different strategies to represent states. A state can be very detailed or more abstract. Selecting the right state representation aids in balancing between "overfitting" (i.e., the model is overspecific 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). Besides selecting the right state representation strategy, it is also possible to further "massage" the transition system using strategies such as "Kill Loops", "Extend", and "Merge by Output". Using the theory of regions, the resulting transition system is transformed into an equivalent Petri net. Also in this phase different settings can be used depending on the desired end-result (using the functionality of Petrify). This makes the approach much more versatile than the approaches described in literature.

The approach has been 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. We also think that by merging the two steps we can develop innovative synthesis methods. 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 develop a "new theory of regions" tailored towards process mining.

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, 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.
3. W.M.P. van der Aalst, H.A. Reijers, and M. Song. Discovering Social Networks from Event Logs. *Computer Supported Cooperative work*, 14(6):549–593, 2005.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. A. Datta. Automating the Discovery of As-Is Business Process Models: Probabilistic and Algorithmic Approaches. *Information Systems Research*, 9(3):275–301, 1998.
14. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.
15. J. Desel and W. Reisig. The Synthesis Problem of Petri Nets. *Acta Informatica*, 33(4):297–315, 1996.
16. 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.

17. 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.
18. 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.
19. B.F. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM framework: A New Era in Process Mining Tool Support. In G. Ciardo and P. Darondeau, editors, *Application and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer-Verlag, Berlin, 2005.
20. 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.
21. A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures - Part 1 and Part 2. *Acta Informatica*, 27(4):315–368, 1989.
22. 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.
23. C. Guenther 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.
24. 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.
25. 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.
26. 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.
27. A.K.A. de Medeiros. *Genetic Process Mining*. PhD thesis, Eindhoven University of Technology, Eindhoven, 2006.
28. 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.
29. 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.
30. 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.
31. 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.
32. 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.
33. 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.
34. TIBCO. TIBCO Staffware Process Monitor (SPM). http://www.tibco.com, 2005.
35. 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.