Mining Structured Petri Nets for the Visualization of Process Behavior [*]

Javier de San Pedro
jspedro@cs.upc.edu

Jordi Cortadella
jordi.cortadella@upc.edu

Universitat Politecnica de Catalunya
Barcelona, Spain

ABSTRACT

Visualization is essential for understanding the models obtained by process mining. Clear and efficient visual representations make the embedded information more accessible and analyzable. This work presents a novel approach for generating process models with structural properties that induce visually friendly layouts. Rather than generating a single model that captures all behaviors, a set of Petri net models is delivered, each one covering a subset of traces of the log. The models are mined by extracting slices of labelled transition systems with specific properties from the complete state space produced by the process logs. In most cases, few Petri nets are sufficient to cover a significant part of the behavior produced by the log.

CCS Concepts

•**Applied computing → Business process modeling;**

Keywords

Petri nets, process mining, visualization

1. INTRODUCTION

Process mining is used to deliver valuable insight into the execution of real-life processes. *Discovery* [21], one of the major areas of process mining, fosters this goal by constructing abstract process models that describe the high level structure of the process. These models are automatically learned from the execution traces of the process. Real-life processes, however, are often highly unstructured. Traces obtained from running systems generally show repetitive patterns, but also contain plenty of unrelated events.

Building unique and compact models out of such unstructured behavior results in the so-called *spaghetti* models, such as the one shown in Fig. 1. As models are generated by automatically learning causal dependencies between different events, fitting many unrelated behaviors into a single model produces highly complex relations. The model may learn fake dependencies between events that are, in truth, unrelated in the original process. These spaghetti models are almost impossible to visualize and understand, and therefore do not deliver any real insight into the underlying process.

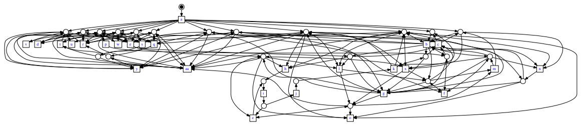


Figure 1: Overfitting "spaghetti" model.

One possible way for simplifying a model is by overgeneralizing the model, i.e. allowing behavior that is not present in the execution traces. Models such as the *flower* Petri net, in which everything is allowed, are conceptually simple, but fail to provide useful information. In fact, it has been argued that the main problem of discovery is finding this balance between *overfitting* and *underfitting* models [19]. It is not easy to satisfy this balance in an automated fashion, and there is plenty of research in the area.

In this paper, we propose a new methodology to mine easy-to-understand process models (Petri nets) from logs, even those containing unstructured data. At the core of this proposal is a new *clustering* approach which automatically separates out behaviors with structural properties that induce visually friendly models. Thereby, our approach generates multiple process models, each of them conceptually simple, but without removing any of the behaviors. For most types of processes only a few models are required to cover a majority of the behavior.

1.1 Related work

The problem of trying to find simple yet fitting models from process logs has been approached from several areas. The closest approaches are those related to clustering. Our approach is novel in that clustering is performed by examining properties of transition systems, not log properties, thus being more accurate in determining which clusters will be visually friendly. A related approach is presented in [9], where the quality and simplicity of the models obtained by the heuristic miner is used to determine similarity of traces. Our approach does not depend on any miner during the clustering process. Most other clustering methods estimate similarity based on the log structure, such as vector distance [18] or edit distance [4]. Significantly, event ordering information can be used to construct the pattern vectors [2]. [10] also targets complexity reduction, but it does so by using hierarchical clustering, unifying repeated patterns.

[*] This work has been partially supported by the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R) and by a grant from Generalitat de Catalunya (FI-DGR).

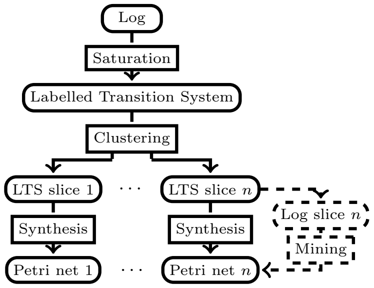


Figure 2: Scheme of the proposed flow.

Another group of techniques try to mine simple nets directly, by selectively ignoring noise from logs, such as the inductive miner [14] or the region-based FSM miner [19]. When two or more significantly distinct behaviors are present in the input logs, these methods must greatly reduce fitness or precision to generate simple models. Our clustering approach, instead, generates separate models for the separate behaviors.

The rest of simplification approaches reduce the complexity of models already mined. For example, the authors of [11] perform a log-based unfolding of the model and compute a new model that only keeps the frequent parts. In [?], an ILP model is also used to simplify existing models while guaranteeing fitness. Like the previous strategies, these approaches also end up generating a single model.

2. METHOD OVERVIEW

In this section we describe the contributions of this work. Figure 2 shows the main components of the mining flow.

The most important step is the clustering process, which separates the log into several (possibly non-disjoint) subsets of traces. The classification criteria is based on exploring properties of the labelled transition system (LTS) constructed from the log. The procedure extracts slices from the LTS induced by subsets of transitions satisfying certain properties. Each LTS slice has an associated subset of traces from the log (a log slice).

By ensuring certain properties on each LTS slice, Petri nets with specific structures can be synthesized. This work focuses on *Marked Graphs* and *Free-Choice nets* because of their inherent visually friendly structure. However, the paradigm does not preclude to use other structural properties that may induce other classes of nets. The details on how the slices are generated are described in Section 4.

In the second step, the different log slices are converted into Petri nets (dashed line in Fig. 2). Any miner can be used to transform these clusters into a Petri net. However, this work proposes a method based on the theory of regions that generates Petri nets based on the LTS slices produced by the clustering step.

2.1 Visual example

Figure 3 illustrates the different stages of the mining flow with a simple example. Fig. 3a contains a fictional event log used as input to the flow. This log is not very complicated, containing only 7 cases and 5 event types.

Fig. 3b shows the LTS constructed from the log, where the presence of transition $s_1 \xrightarrow{d} s_{11}$, coloured in Fig. 3b, violates certain structural properties (later detailed in Section 4). These violations imply that it is not possible to synthesize a visually-friendly Petri net from the LTS. The result is the intricate structure shown in Fig. 3c.

The clustering process proposed in this paper finds a set of slices of the LTS that satisfy certain structural properties. In this case, the LTS only needs to be split into two slices, shown in Fig. 3d. Each slice also fits only a subset of the traces from the log, shown in Fig. 3e, and these log slices are the result of the clustering process. In this particular case, the log slices are disjoint. However, overlapping slices can be produced in the most general case.

Figure 3f shows the Petri nets synthesized from the individual LTS slices. The LTS properties give rise to free-choice Petri nets, which are much more visually friendly.

3. BACKGROUND

3.1 Process Mining

Process models are formalisms to represent the behavior of a process. Among the different formalisms, Petri nets are perhaps the most popular, due to its well-defined semantics.

A *trace* is a word $\sigma \in T^*$ that represents a finite sequence of events. An *event log* $L \in \mathcal{B}(T^*)$ is a multiset of traces[1]. Event logs are the starting point to apply process mining techniques, guided towards the discovery, analysis or extension of process models. *Process discovery* is one of the most important disciplines in process mining, concerned with learning a process model (e.g., a Petri net) from an event log. Although a novel discipline, there are several discovery techniques that have appeared in the last decade, most of them summarized in [21].

For any trace $\sigma \in T^*$, the Parikh vector $\psi(\sigma)$ maps every element $a \in T$ onto the number of occurrences of a in σ.

3.2 Petri nets

A Petri Net [16] is a tuple $N = \langle P, T, \mathcal{F}, m_0 \rangle$, where P is the set of places, T is the set of transitions, connected via the flow relation $\mathcal{F} : (P \times T) \cup (T \times P) \rightarrow \{0, 1\}$, and m_0 is the initial marking. A marking is an assignment of a non-negative integer to each place. If k is assigned to place p by marking m (denoted $m(p) = k$), we say that p is marked with k tokens. Given a node $x \in P \cup T$, its pre-set and post-set are denoted by ${}^\bullet x$ and $x^\bullet$ respectively.

A transition t is *enabled* in a marking m when all places in ${}^\bullet t$ are marked. When t is enabled, it can *fire* by removing a token from each place in ${}^\bullet t$ and putting a token to each place in $t^\bullet$. A marking m' is *reachable* from m if there is a sequence of firings $t_1 t_2 \ldots t_n$ that transforms m into m', denoted by $m[t_1 t_2 \ldots t_n\rangle m'$. A sequence $t_1 t_2 \ldots t_n$ is *feasible* if it is firable from m_0.

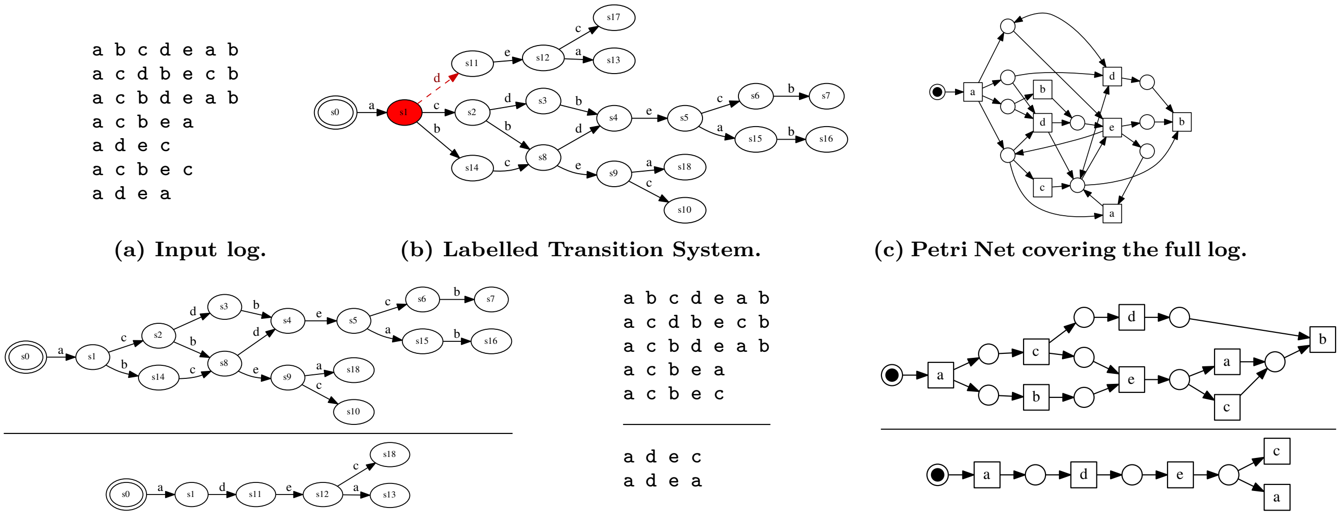
In a Petri net, a *choice* is a place with more than one output transition. Two transitions are said to be *concurrent* if they do not have dependencies between them, i.e. they can fire in any order.

A set of restrictions on the structure of Petri nets define several classes of Petri nets. A Petri net N is a *Marked Graph* if $\forall p \in P : |{}^\bullet p| = |p^\bullet| = 1$. It is a *Free-Choice* net if $\forall p_1, p_2 \in P : p_1^\bullet \cap p_2^\bullet \neq \emptyset \;\Rightarrow |p_1^\bullet| = |p_2^\bullet| = 1$. Note that every marked graph is a free-choice net.

3.3 Labelled Transition Systems

A finite labeled trasition system is a tuple $A = (S, E, T, s_0)$ where S is a finite set the states, E

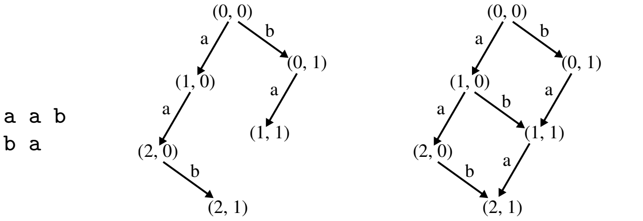
[1] $\mathcal{B}(A)$ denotes the set of all multisets over A.



(a) Input log. (b) Labelled Transition System. (c) Petri Net covering the full log.

(d) Two slices extracted from the LTS. (e) Corresponding log-slices. (f) Synthesized Petri nets.

Figure 3: Examples at different stages of the mining flow.



(a) Log. (b) *Prefix multiset* LTS. (c) Arc-completed LTS.

Figure 4: Log and steps to construct LTS.

is a set of events, $T \in S \times E \times S$ are the transition relations between states, labelled with E, and s_0 is the initial state.

We use $s \xrightarrow{e} s'$ as a shorthand for the arc $(s, e, s') \in T$. Similar to Petri nets, a sequence $\sigma = e_1 e_2 \ldots e_n$ is *feasible* in A if there exists $s_1, s_2, \ldots, s_n \in S$ with $s_0 \xrightarrow{e_1} s_2 \xrightarrow{\cdots} s_{n-1} \xrightarrow{e_n} s_n$.

For a given LTS $A = (S, E, T, s_0)$ and event $e \in E$, we define the *Excitation Set* of e as the set of states in which e is enabled, i.e.,

$$ES(e) = \{s \in S \mid \exists s' \in S : s \xrightarrow{e} s'\}.$$

Two events a, b are *concurrent* if there are four states, $s_1 \ldots s_4$, in S such that $s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_4$ and $s_1 \xrightarrow{b} s_3 \xrightarrow{a} s_4$. In this case we will also say that a and b are concurrent in s_1. Two events a, b are in *conflict* if there is a state $s \in ES(a) \cap ES(b)$ and a, b are not concurrent in s.

The previous definitions allow two events to have multiple relationships in the same LTS (concurrency, conflict or ordered). One of the goals of this work would be to find LTS slices with "clean" relationships that can be easily visualized in the form of Petri nets.

Next, a concept tightly related to Free-Choice nets is presented. Two events a and b are in *free-choice conflict* if they are in conflict and $ES(a) = ES(b)$. In this situation the two events are always enabled or disabled simultaneously, which corresponds to a similar situation in Free-Choice nets.

3.4 Construction of an LTS from a log

Many methods exist to construct an LTS from a log. This work uses a variation of the *prefix multiset* conversion [19].

In real-life, logs obtained from execution traces often miss information required to fully learn the correct process model. For example, in the log in Fig. 4a, events a and b are seemingly concurrent as a is still enabled after firing b and viceversa. The log shows trace $b\,a$, but does not contain $a\,b$. The original prefix multiset technique would generate an LTS where a and b are not concurrent (Fig. 4b). The technique presented below generates an LTS where $a\,b$ is also feasible.

For an input log $L \in \mathcal{B}(T^*)$, the procedure creates a new LTS A as follows:

1. The Parikh vector $\psi(\sigma)$ of every possible prefix $\sigma \in T^*$ appearing in L is computed. A new state is created for every such different Parikh vector.
2. An arc $s_1 \xrightarrow{e_i} s_2$ is created for every pair $s_1, s_2 \in A$ if $\psi(s_1) = (x_0, \ldots, x_i, \ldots, x_n)$ and $\psi(s_2) = (x_0, \ldots, x_i + 1, \ldots, x_n)$.
3. The initial state s_0 is the zero Parikh vector.

An example of a log and the LTS generated using this method can be seen in Fig. 4c.

3.5 Conformance checking

An important set of techniques in process mining is *conformance checking*, which compare the observed (log) and modeled behavior in order to evaluate the model. There are four quality dimensions for comparing model and log: *replay fitness*, *simplicity*, *precision*, and *generalization* [21].

A trace $\sigma \in T^*$ *fits* a model N if σ is a feasible sequence in N. A model has a perfect replay fitness if all traces in the log can be replayed by the model. A precise model, on the other hand, does not replay any trace other than those contained in the log. Metrics for fitness have been defined indicating how much every trace from the log fits a model [1]. Likewise, metrics for precision exist in the literature [17].

4. CLUSTERING

This section explains the main contribution of this work: a clustering procedure to extract LTS slices that cover subsets of the log. The extraction is done in such a way that each log slice can be modeled by a Petri net with visually friendly structure (free-choice) that covers the log slice with 100% fitness. Clustering is done in two steps:

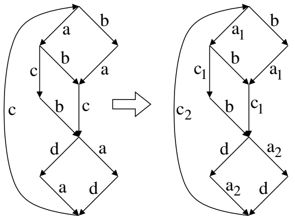


Figure 5: Splitting maximal connected ESs.

1. Transforming the LTS to enforce specific relations between pairs of events.
2. Extracting LTS slices with pre-defined structural properties.

The first step (explained in Section 4.1) transforms the LTS to enforce unique relationships between pairs of events. After these transformations, all pairs of events in the LTS are guaranteed to have only one relationship: ordered, in conflict or concurrent.

The second step extracts slides from the LTS in such a way that certain properties are met. These properties are related to the state spaces of Marked Graphs [3] and Free-Choice nets [8]. The extraction will be based on a satisfiability model that characterizes all slices that meet the required properties (see Section 4.2).

4.1 LTS transformations

The transformations presented in this section aim at enforcing unique concurrency/conflict relationships between pairs of events.

4.1.1 Splitting events with disconnected ESs

It is frequent to observe disconnected instances of the same event in different states of the LTS. This transformation considers each instance as a different event.

The transformation splits each event e into a set of events $e_1 \dots e_k$ such that the ESs of each event is maximally connected (only has one connected component). Figure 5 illustrates this transformation with an example, in which event a has two maximally connected sets of states in $ES(s)$. Therefore, two new events, a_1 and a_2 are created to substitute the original event a. A similar situation occurs with event c.

When synthesizing a Petri net from the LTS, these events may end up by being represented by different transitions in the Petri net if the Free-Choice property requires the splitting. The decision about whether the splitting is necessary is left at the criterion of the synthesis tool [8].

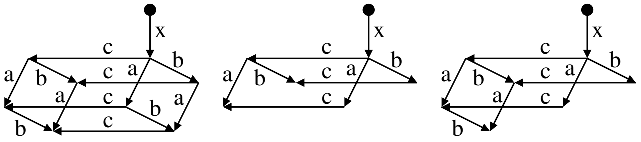
4.1.2 Saturation of concurrency

The goal of this transformation is to define a unique relationship between every pair of events. For every pair a and b, there are two options:

$$ES(a) \cap ES(b) = \emptyset \quad \Rightarrow \quad \text{ordered.}$$
$$ES(a) \cap ES(b) \neq \emptyset \quad \Rightarrow \quad \text{concurrent and/or conflict.}$$

This transformation aims at disambiguating the second case by exclusively choosing between concurrency or conflict. The following definition formally describes what concurrency saturation is.

Two concurrent events are *concurrency-saturated* (c-saturated) if they are concurrent in all states of $ES(a) \cap ES(b)$. An LTS is said to be c-saturated if all pairs of events are c-saturated.



(a) (b) (c)

Figure 6: Examples to illustrate c-saturation.

Figure 6 shows three LTSs with different concurrency properties. The one in 6a is c-saturated since all pairs of concurrent events, (a, b), (b, c), and (a, c), are c-saturated. The one in 6b is also c-saturated with the pairs (a, c) and (b, c) being concurrent. However, 6c is not c-saturated since the pair (a, b) is concurrent, but not in all the states of $ES(a) \cap ES(b)$. Therefore the pair (a, b) is also in conflict.

Any LTS can be transformed into c-saturated by adding states and transitions to complete the missing diamonds of the concurrent events. This process can be applied iteratively until reaching a fixpoint in which all pairs of concurrent events are c-saturated. The details of the algorithm are not given in the paper, but the reader can easily figure out how to design it and verify that converges to a finite LTS.

In the example of Fig. 6, the LTS in 6c would become the one in 6a after c-saturation. With this transformation, there is no pair of events that can be concurrent and in conflict simultaneously. This contributes to remove intricate event relations, thus resulting in simpler Petri net structures.

4.2 Extraction of LTS slices

Once an LTS A representing the input log L is constructed and transformed, the following step extracts several LTS slices $A_1, A_2, \dots, A_k$ satisfying a set of properties than make it amenable for the synthesis of visually friendly Petri nets. Each slice A_i covers a subset of traces (log-slice) L_i of L. The output of the clustering process is a set of log-slices that completely cover L. In the example of Fig. 3, two log-slices are delivered, shown in Fig. 3e.

We first describe the properties enforced in the LTS slices and then the satisfiability model that extracts the slices.

4.2.1 Properties of the LTS slices

Three properties are desired in the LTS slices: forward persistency, backward persistency and free-choiceness.

Persistency is a property tightly related to the state spaces of Marked Graphs (see [3]). An LTS $A = (S, E, T, s_0)$ is *forward persistent* (FP) if

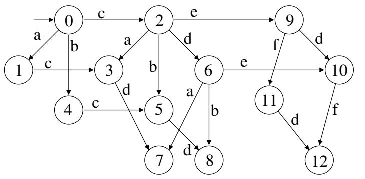
$$\forall s_1 \xrightarrow{a} s_2, s_1 \xrightarrow{b} s_3 : \quad \exists s_3 \xrightarrow{a} s_4.$$

Informally, if two events are simultaneously enabled, they must be concurrent. *Backward persistency* (BP) is an analogous property applied to the reversed LTS (reversing the direction of transitions). It is known that forward and backward persistency are necessary conditions for an LTS to be the state space of a Marked Graph [3].

The third property is free-choiceness (FC). An LTS is said to be free-choice if for every pair of events a and b, the following condition holds:

$$a \text{ and } b \text{ are in conflict} \quad \Longrightarrow \quad ES(a) = ES(b).$$

FC characterizes the state space for conflicts in Free-Choice Petri nets. Given that two transitions in conflict have the same predecessor places in a free-choice net, the set of markings in which they are enabled is identical for both transi-



	b	c	d	e	f
a	F	∥	∥	C	S
b		∥	∥	C	
c					
d				∥	∥
e					

Figure 7: LTS with various conflict relations between events.

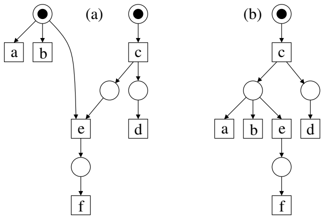


Figure 8: Petri nets obtained from the synthesis of the LTS in Figure 7: (a) from the the full LTS, (b) from a slice obtained by removing states 1 and 4.

tions. This means that both excitation sets will be identical in the corresponding LTS. The FC property is the one used in [8] to split events and guarantee free-choiceness.

Figure 7 shows an LTS and the relationship between pairs of events (∥: concurrent, C: conflict, F: free-choice conflict). Blank cells in the table represent ordered events (disjoint ES's). We observe that the only free-choice conflict is between a and b, given that $ES(a) = ES(b) = \{s_0, s_2, s_6\}$. The conflicts for the pairs (a, e) and (b, e) are not free-choice.

Figure 8 depicts two Petri nets obtained from the LTS in Fig. 7. The one in Fig. 7a has been obtained by synthesizing the full LTS, whereas the one in Fig. 7b has been obtained by a slice in which states 1 and 4 have been removed. In the latter case, all conflicts for the pairs (a, b), (a, e) and (b, e) become free-choice, and so does the Petri net.

4.2.2 Satisfiability model

An LTS slice is simply a subset of the original LTS. The goal of the approach presented in this paper is to extract LTS slices with the FP, BP and FC properties. Let us call them *Well-Behaved (WB) slices.*

If each transition $t_i \in T$ of the LTS is represented by a Boolean variable, the set of WB-slices can be characterized by a Boolean formula $\mathrm{WB}(T)$ in which every satisfying assignment corresponds to a WB-slice that contains only the transitions t_i for which their variables are asserted.

Fortunately, the function $\mathrm{WB}(T)$ can be easily constructed by observing that the FP, BP and FC properties can be formulated locally, i.e., in terms of neighboring transitions. Once the WB formula is constructed, a SAT solver can be used to extract slices. Alternatively, the SAT model can be transformed into an Integer Programming model in which some specific cost function can be optimized.

Let us now see how the FP, BP and FC properties can be characterized with Boolean constraints. Note that after the transformations applied to the LTS, all pairs of events have a unique relationship, i.e., they can only be concurrent or in conflict (but not both) in case their ESs intersect.

Forward persistency (FP).

This property is only applicable to pairs of concurrent events (a, b). For every state $s_1 \in ES(a) \cap ES(b)$, a diamond exists with the following transitions:

$$t_1 = (s_1, a, s_2), t_2 = (s_1, b, s_3), t_3 = (s_2, b, s_4), t_4 = (s_3, a, s_4).$$

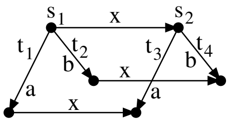


Figure 9: Example to illustrate the FC constraint.

Any selected subset of $\{t_1, t_2, t_3, t_4\}$ must preserve the FP property, which means that the selection of t_1 and t_2 must imply the full diamond. The constraint can be formulated as follows:

$$(t_1 \wedge t_2) \implies (t_3 \wedge t_4).$$

Backward persistency (BP).

This property is analogous to the previous one, but reversing the direction of the transitions. A similar Boolean formulation can be constructed.

Free-choiceness (FC).

This property is applied to pairs of events in conflict. The constraints must ensure that both events have the same excitation sets in case both events are present in the LTS slice.

The formalization of the constraints for each pair of events in conflict can be stated as follows:

> Once one of the events is enabled in one state, then the same enabling relation must be maintained in the successor states reachable by other events.

Figure 9 depicts an scenario in which FC must be applied for two events, a and b, that are in conflict. Notice that the event x with transition $s_1 \xrightarrow{x} s_2$ must be concurrent with a and b, otherwise a, b would not be enabled in s_2, according to the c-saturation transformation applied to the LTS.

Three options are possible with regard to the selection of transitions t_1 and t_2 at state s_1:

1. Both t_1 and t_2 are selected: in this case, the two events must be enforced to have the same ESs and, thus, both or none of t_3 and t_4 must be selected, i.e., $t_3 \Leftrightarrow t_4$.
2. Only one of them is selected, say t_1. In this case, t_3 would be present (because of the FP condition) and t_4 cannot be selected, otherwise $ES(a) \neq ES(b)$.
3. None of them is selected. In this case, no constraints are imposed on t_3 and t_4 with regard to state s_1.

Formally, the Boolean constraint applicable to s_1 with regard to events a and b is stated as follows:

$$\begin{aligned}(t_1 \wedge t_2) &\implies (t_3 \Leftrightarrow t_4)\\ (t_1 \wedge \neg t_2) &\implies \neg t_4\\ (\neg t_1 \wedge t_2) &\implies \neg t_3\end{aligned}$$

The previous constraints can be simplified in case a and b are in conflict but not all transitions of Fig. 9 are present in the original LTS. For example, if t_1 is not present ($\neg t_1$), the previous constraints would be simplified and reduced to:

$$t_2 \implies \neg t_3.$$

Generation of choice-free Petri nets.

The previous constraints characterize LTS slices that derive Petri nets in which all choices are free. A simple modification of the model can characterize Petri nets without choices (only causality and concurrency relations). In particular, the FC conditions can be rewritten to disable the selection of two transitions that are in conflict. This would be equivalent to adding the constraint $\neg t_1 \vee \neg t_2$.

4.3 Trace coverage

The conjunction of constraints for FP, BP and FC conform the Boolean formula $\mathrm{WB}(T)$ that characterizes all WB-slices of the LTS. The question now is: which subset of slices should be extracted? Two properties are desired:

- Every trace of the log must be covered by at least one WB-slice.
- A small number of WB-slices should cover the majority of traces of the log.

At this point, the satisfiability problem becomes an optimization problem that can be modeled with an Integer Programming model with only binary variables.

A log L is a set of traces $L = \{\sigma_1, \ldots, \sigma_n\}$, and each trace σ_i covers a set of transitions $\sigma_i = \{t_{i_1}, \ldots, t_{i_k}\}$ of the LTS, according to the construction described in Section 3.4.

Each trace can be represented by a Boolean variable σ_i that acts as a trace selector. The assertion of σ_i implies the selection of all transitions of the trace in the LTS, i.e.,

$$\sigma_i \implies (t_{i_1} \wedge \cdots \wedge t_{i_k})$$

Including this constraint in the model, a subset of traces can be covered by selecting their variables. The number of covered traces is maximized by incorporating a cost function:

$$\max \sum \sigma_i$$

4.4 Algorithm for extracting WB-slices

Algorithm 1 Extraction of WB-slices

```
 1: Input: A log $L$
 2: Output: A set of pairs (LTS slice, Log slice)
 3: $A \leftarrow$ c-saturated LTS from $L$
 4: $R \leftarrow L$                    ▷ Remaining (uncovered) traces from $L$
 5: $i \leftarrow 1$
 6: while $|R| > 0$ do
 7:     $A_i \leftarrow$ SolveWB$(A, R)$            ▷ extract new LTS slice
 8:     $T_i \leftarrow$ traces from $L$ fitting $A_i$      ▷ associated traces
 9:     $R \leftarrow R \setminus T_i$             ▷ subtract from remaining traces
10:     $i \leftarrow i + 1$
11: return $(A_1, T_1), (A_2, T_2), \ldots, (A_n, T_n)$
```

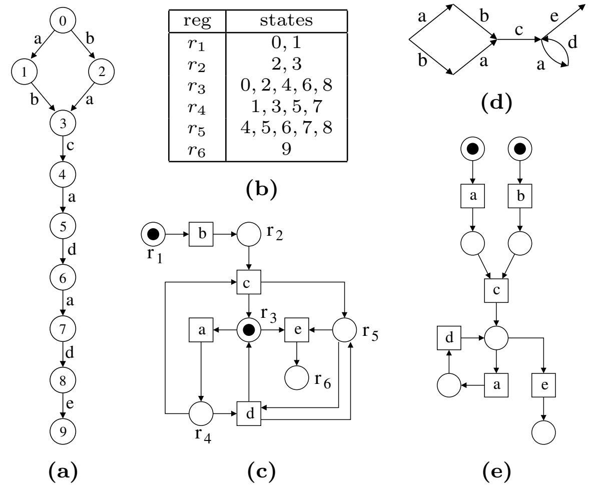
Algorithm 1 shows the main algorithm for extracting LTS and log slices. The algorithm iterates until each trace in log L is covered by at least one of the extracted LTS slices. Variable R stores the list of traces not yet covered.

Function SolveWB extracts an slice from A that maximizes the number of covered traces from R. Still, this slice may cover traces already covered by previous slices (in line 8, the traces assigned to T_i are obtained from L, not R).

5. SYNTHESIS OF PETRI NETS

In addition to the main clustering procedure, we propose a method to transform the LTS slices into Petri nets, extending a region-based synthesis tool, *Petrify* [7]. Region-based miners are known for their tendency to construct overfitting models [21]. The proposed modification allows Petrify to construct Petri nets that trade off precision for visualization-friendliness, and implements the ideas of [5].

Any mining tool can be used to obtain process models from the log slices generated by the clustering method. However, the synthesis procedure described in this section reuses the LTS slices constructed by the clustering algorithm, and thus has inherent benefits in both efficiency and simplicity.



reg	states
r_1	$0, 1$
r_2	$2, 3$
r_3	$0, 2, 4, 6, 8$
r_4	$1, 3, 5, 7$
r_5	$4, 5, 6, 7, 8$
r_6	9

Figure 10: Synthesis of Petri nets using regions.

For full details on the theory of regions, we refer to [8]. The following section briefly surveys the basics of the theory.

5.1 Theory of regions

The theory of regions provides an algorithmic method to derive a Petri net from an LTS. Given an LTS $A = (S, E, T, s_0)$ and a subset of states $r \subseteq S$, a transition $s \xrightarrow{e} s'$ *enters* r if $s \notin r \wedge s' \in r$, *exits* r if $s \in r \wedge s' \notin r$, and *does not cross* r otherwise. r is a *region* if for every event $e \in E$, all transitions $t \in T$ labelled e are in the same relation with r (entering, exiting, or not crossing). A region r is *minimal* if no subset of r is a region. Figure 10a shows an example LTS, with 10b listing the minimal regions.

Intuitively, a region r corresponds to a place p in the Petri net. p will precondition any of the events exiting r. Thus, it will be marked only in states corresponding to those in r, receiving tokens only from transitions corresponding to events entering r. Straightforwardly, to construct a Petri net from the list of minimal regions, a transition t_i is created for every event $e_i \in A$ and a place p_j is created for every minimal region r_j. If e_i enters r_j, then $p_j \in {}^\bullet t_i$. Conversely, if e_i exits r_j, then $p_j \in t_i^\bullet$. p_j is marked iff $s_0 \in r_j$.

5.2 Excitation Closure

Let $\mathcal{L}(A)$ be the *language* of the LTS A, i.e., the set of all traces fitting A, and $\mathcal{L}(N)$ the set of all traces fitting N, where N is a Petri net constructed from A using the above method. Assuming $e \in E$, let ${}^\circ e$ be the set of minimal regions of A where e is exiting. A is *excitation closed* [8] if:

$$\forall e \in E,\ \ {}^\circ e \neq \emptyset \ \wedge\ ES(e) = \bigcap_{r \in {}^\circ e} r$$

$\mathcal{L}(A) = \mathcal{L}(N)$ is only guaranteed if A is excitation closed [8]. If not, then $\mathcal{L}(A) \subseteq \mathcal{L}(N)$ [5], N fits more traces than A.

Figure 10c depicts the Petri net N obtained from the minimal regions of the LTS A shown in 10a, whereas 10d shows the LTS that models $\mathcal{L}(N)$. Since the LTS in 10a is not excitation closed, N fits more traces than A (e.g., trace abce).

Traditional region-based synthesis tools, including Petrify, transform the LTS to guarantee the excitation closure property and thus enforce $\mathcal{L}(A) = \mathcal{L}(N)$. These transformations often degrade the visualization quality of the resulting Petri nets [5] and result in hard to understand overfitting models which are not ideal for Process Mining. For these reasons,

Table 1: Time required for clustering.

	Log size		Time [s.]	
Log	Uniq. traces	Event types	Our proposal	ActiTraC
documentflow	1411	70	90	95
fhmilu	701	12	60	106
fhmn5	693	13	174	38
incidenttelco	212	22	32	17
kim	1174	18	37	20
purchasetopay	76	21	7	55
receipt	116	27	15	7
tsl	1908	42	111	60

Table 2: Number of slices and crossings required to reach specific fitness thresholds.

Our proposal						
Log	85%		90%		95%	
	Slic.	Cros.	Slic.	Cros.	Slic.	Cros.
documentflow	1	0	1	0	4	8
fhmilu	1	4	1	4	1	4
fhmn5	1	1	1	1	1	1
incidenttelco	1	0	1	0	2	1
kim	1	0	1	0	2	0
purchasetopay	1	0	1	0	1	0
receipt	1	3	1	3	1	3
tsl	1	3	3	3	10	9

ActiTraC						
Log	85%		90%		95%	
	Slic.	Cros.	Slic.	Cros.	Slic.	Cros.
documentflow	1	0	2	14	3	946701
fhmilu	6	4	*Timeout computing fitness*[2]			
fhmn5	3	1	6	2	8	2
incidenttelco	1	3	1	3	2	3
kim	1	2	1	2	2	3
purchasetopay	1	2	1	2	1	2
receipt	2	5	2	5	2	5
tsl	1	25	1	25	2	145

our modification drops the excitation closure requirement to avoid these transformations. Because of this relaxation, the resulting Petri nets are less overfitting.

Finally, event splitting can contribute to further enhance the structure of the Petri net while keeping the recognized language. Figure 10e shows a new structure after splitting event a. The details on how label splitting is performed are out of the scope of this work (see [8]).

6. RESULTS

We have implemented the algorithms described on this work in Python, using the PMLAB [6] process mining package. 8 logs combining real-life sources and benchmarks have been used as inputs to the mining process. These logs are available at http://www.cs.upc.edu/~jspedro/pnsimpl/. Gurobi [13] has been used to solve all ILP models. We have evaluated the quality of the obtained models using the ET-Conformance plugin in ProM, which compute the best alignment between trace and log before measuring fitness and precision [1, 17].

As for the important issue of visualization-friendliness, there are several studies in the literature [15]. In this work we propose a metric related to the concept of planarity: the minimal number of crossings required to embed the graph on a plane. This value is hard to compute optimally. For this reason, the *mincross* algorithm from Graphviz [12] is used to obtain the estimations that will be used in this section. As it is only an estimation, *mincross* may sometimes overestimate the number of crossings in large, dense graphs. These pathological cases, most often, are not visualization-friendly even if the crossing number is overestimated.

In order to compare our proposal with the related work we propose two experiments. In the first one, we show how many slices (Slic.) and crossings (Cros.) are required by our mining process, depending on how much behavior of the log is preserved. We also compare it to *ActiTraC* [9], a state-of-the-art clustering procedure that also targets visualization.

In the second experiment, we show how our slicing approach results into visualization-friendly models even when using alternative miners.

6.1 Minimum number of log slices

We configured our implementation to generate as many slices as required in order to obtain a set of slices that include at least 80% of traces from the input log. Petrify was used to generate models for each of the slices. The size of the logs, in number of unique traces and event types, as well as the time spent during the slicing process is shown in Table 1. Similarly, *ActiTraC* was configured with the default settings, but a stop criteria of having generated enough clusters to cover 80% of the traces. By default, ActiTraC uses the Heuristic Miner to generate models for the clusters.

The models mined by both tools are generally underfitting, i.e., precision is sacrificed to aid model visualization. Thus, each model allows for more traces than those present in the corresponding log slices. For this reason, measuring entire log replay fitness using ProM usually results in values higher than 80% of the log as configured.

In this experiment, we measure how many slices are actually required to reach a specific level of fitness when replaying the entire input log. For both clustering algorithms, the slices with the largest number of traces are selected first. This number of slices estimates how much behavior a clustering algorithm can fit into a single model. However, the clustering algorithm needs to ensure each model is still visualization-friendly. To estimate this, we measure the total number of crossings present in each of the selected slices.

The results, shown in Table 2, indicate that our approach compares positively with ActiTraC. For most examples, the first slice already provides with a model that fits 90% of the original full log. In addition, these first slices have few or almost no crossings.

6.2 Metrics of the first slice

In the second experiment, we repeat the proposed slicing procedure on the same logs, but center on the metrics of the slice containing the highest number of traces, which we call the *first slice*. Two distinct miners are used: Petrify (using the modifications described in Section 5) and the α-miner [20]. The experiment shows how the slices generated by the proposed algorithm are visualization friendly even when using other types of miners.

As a baseline for comparisons, we also show the metrics of models obtained directly from the log, without applying our slicing approach, but after applying a *naive* noise filtering algorithm. This naive strategy removes the least frequent 20% traces from each log, effectively removing most infrequent behavior. Without this noise removal, attempting to mine the full log would result in huge spaghetti models which

[2] Because of the large number of Petri Nets, replay fitness could not be measured for this benchmark.

Table 3: Comparing the first slice from the proposed clustering method vs. a naive noise removal algorithm (removing 20% least frequent traces), using different miners.

	Using Petrify as miner						Using α miner			
	1st slice (our method)			Naive method			1st slice (our)		Naive method	
Log	**Fit.**	**Prec.**	**Cros.**	**Fit.**	**Prec.**	**Cros.**	**Fit.**	**Cros.**	**Fit.**	**Cros.**
documentflow	92.1%	81.7%	0	97.9%	57.2%	0	37.8%	0	37.5%	2286
fhmilu	97.8%	64.5%	4	*Out of memory*			38.0%	8	16.8%	29
fhmn5	99.1%	49.4%	1	*Out of memory*			40.6%	4	25.5%	116
incidenttelco	92.6%	53.7%	0	99.3%	30.7%	14	63.1%	9	46.7%	53
kim	92.9%	75.1%	0	94.6%	84.6%	38	66.4%	7	56.1%	80
purchasetopay	99.8%	68.2%	0	94.0%	100.0%	0	96.9%	6	100.0%	0
receipt	98.0%	71.3%	3	95.9%	100.0%	0	99.8%	8	76.6%	1
tsl	82.7%	83.9%	3	99.7%	53.1%	1	*Timeout*		75.7%	310

would be meaningless to compare with the models generated after our slicing procedure. Yet, the results show that even when compared to models where most noise has been removed, using our slicing algorithm still results in simpler models with comparable fitness (Fit.) and precision (Prec.).

Table 3 shows these metrics on the first slice of each of the logs as well as using the naive algorithm. When using Petrify as miner, some of the models generated after the naive noise elimination still have tens of crossings or low precision. The models generated all have very few crossings despite maintaining similar levels of fitness and precision.

When using the α-miner, low-fitting spaghetti models are discovered even after noise elimination. When applied to the first slices, the α-miner discovers models with a significantly reduced number of crossings, and increased fitness.

7. CONCLUSIONS

In this paper, we have introduced a new method to mine visualization-friendly Petri nets from real-life process logs. We have shown our method to improve on the visualization quality of other similar slicing methods while being competitive in performance. This work is just an initial incursion into the study of properties of labelled transition systems with the goal of process model visualization. We envision this effort to be a starting point for further simplifications using other structural properties.

8. REFERENCES

[1] A. Adriansyah. *Aligning observed and modeled behavior.* PhD, Technische Universiteit Eindhoven, 2014.

[2] A. Appice and D. Malerba. A co-training strategy for multiple view clustering in Process Mining. *IEEE Trans. on Services Computing*, 2015. Early online access.

[3] E. Best and R. Devillers. Characterisation of the state spaces of live and bounded marked graph Petri Nets. In *Language and Automata Theory and Applications*, volume 8370 of *LNCS*, pages 161–172. Springer, 2014.

[4] R. P. J. C. Bose and W. M. P. van der Aalst. Context aware trace clustering: Towards improving process mining results. In *Proceedings of the 2009 SIAM International Conference on Data Mining*, pages 401–412, 2009.

[5] J. Carmona, J. Cortadella, and M. Kishinevsky. A region-based algorithm for discovering Petri Nets from event logs. In *Business Process Management*, volume 5240 of *LNCS*, pages 358–373. Springer, 2008.

[6] J. Carmona and M. Solé. PMLAB: An scripting environment for Process Mining. In *Proceedings of the BPM Demo Sessions 2014*, pages 16–21, Oct. 2014.

[7] 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*, 80(3):315–325, 1997.

[8] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri nets from Finite Transition Systems. *IEEE T. on Comp.*, 47(8):859–882, Aug. 1998.

[9] J. De Weerdt, S. vanden Broucke, J. Vanthienen, and B. Baesens. Active trace clustering for improved process discovery. *IEEE Trans. on Knowledge and Data Engineering*, 25(12):2708–2720, Dec. 2013.

[10] C. C. Ekanayake, M. Dumas, L. García-Bañuelos, and M. La Rosa. Slice, mine and dice: Complexity-aware automated discovery of business process models. In *Business Process Management*, volume 8094 of *LNCS*, pages 49–64. Springer, 2013.

[11] D. Fahland and W. van der Aalst. Simplifying mined process models: An approach based on unfoldings. In *Business Process Management*, volume 6896 of *LNCS*, pages 362–378. Springer, 2011.

[12] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Trans. Software Eng.*, 19(3):214–230, 1993.

[13] Gurobi Inc. *Gurobi Optimizer Reference Manual.* 2013.

[14] S. Leemans, D. Fahland, and W. van der Aalst. Discovering block-structured process models from event logs - a constructive approach. In *Application and Theory of Petri Nets and Concurrency*, volume 7927 of *LNCS*, pages 311–329. Springer, 2013.

[15] J. Mendling, G. Neumann, and W. van der Aalst. Understanding the occurrence of errors in process models based on metrics. In *On the Move to Meaningful Internet Systems*, volume 4803 of *LNCS*, pages 113–130. Springer, 2007.

[16] T. Murata. Petri Nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, Apr. 1989.

[17] J. Muñoz-Gama and J. Carmona. A fresh look at precision in process conformance. In *Business Process Management*, volume 6336 of *LNCS*, pages 211–226. Springer, 2010.

[18] M. Song, C. Günther, and W. van der Aalst. Trace clustering in Process Mining. In *Business Process Management Workshops*, volume 17 of *LNBIP*, pages 109–120. Springer, 2009.

[19] W. van der Aalst, V. Rubin, H. Verbeek, B. van Dongen, E. Kindler, and C. Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software & Systems Modeling*, 9(1):87–111, 2010.

[20] W. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: discovering process models from event logs. *IEEE Trans. on Knowledge and Data Engineering*, 16(9):1128–1142, Sept. 2004.

[21] W. M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes.* Springer, 1st edition, 2011.