# Region-based Foldings in Process Discovery

Marc Solé, Josep Carmona
E-mail: msole@ac.upc.edu, jcarmona@lsi.upc.edu

**Abstract**

A central problem in the area of Process Mining is to obtain a formal model that represents the processes that are conducted in a system. If realized, this simple motivation allows for powerful techniques that can be used to formally analyze and optimize a system, without the need to resort on its semi-formal and sometimes inaccurate specification. The problem addressed in this paper is known as Process Discovery: to obtain a formal model from a set of system executions.

The theory of regions is probably the most successful approach for process discovery: it aims at learning a formal model (Petri nets) from a set of traces. On its genuine form, the theory is applied on an automaton and therefore one should convert the traces into a tree-like automaton in order to apply these techniques. Given that the complexity of the region-based techniques depends on the size of the input automata, revealing the underlying cycles and folding accordingly the initial automaton can incur in a significant complexity alleviation of the region-based techniques. In this paper we follow this idea by incorporating region information in the cycle detection algorithm, enabling the identification of complex cycles that cannot be obtained efficiently with state-of-the-art techniques. The experimental results obtained by the devised tool suggest that the techniques presented in this paper are a big step into widening the application of the theory of regions in Process Mining for industrial scenarios.

## I. Introduction

The search for global patterns that can be used to make predictions about the future has been one of the key elements that have brought Data Mining to be one of the most relevant research areas in the last decades. Data mining techniques can be applied naturally on large amount of data like databases or even the Internet, and with the help of other disciplines like statistics or machine learning, can effectively reveal important patterns in many scenarios (health care, business, transportation, etc ...).

As in Data Mining, Process Discovery tries to reveal patterns. However, the patterns aimed by Process Discovery techniques are process models, i.e., formal representations of the processes of a system. Due to its different focus, the Process Discovery techniques apply disciplines different from the ones used in data mining, to allow for the derivation of both the statics and the dynamics of a system process. Depending on the emphasis, different dimensions can be considered ranging from social (the identification of communities) [1] to control-flow (the identification of the complex interplay between system's tasks) [2]. In this work we consider the latter: discover a Petri net from a set of traces corresponding to executions of a system (called *Log*).

The first method to obtain a Petri net from a log was presented in [2]. The method (called $\alpha$-*algorithm*) was based on detecting the causal relations in the traces and constructing the Petri net accordingly. The algorithm derived nets within a particular class of Petri nets called *Workflow Nets*, and the technique proved to be a very elegant and light manner to obtain a Petri net from a log. However, the restrictive class of Petri nets that one can obtain by using the $\alpha$-*algorithm* limits its application to very restricted behaviors. To overcome this limitation, several extensions have been presented in the literature to widen the class of Petri nets that the algorithm can discover [3], [4]. Also, a genetic approach was presented in [5], which was able to extend the class of Petri nets obtained but which was unsatisfactory for handling industrial examples due to its inherent complexity.

An alternative approach for discovering Petri nets can be devised by using the *theory of regions* [6]. Importantly, in contrast with the approaches described above, this approach is not restricted to a particular subclass of nets. The theory of regions was initially proposed to solve the *synthesis problem*: obtain a

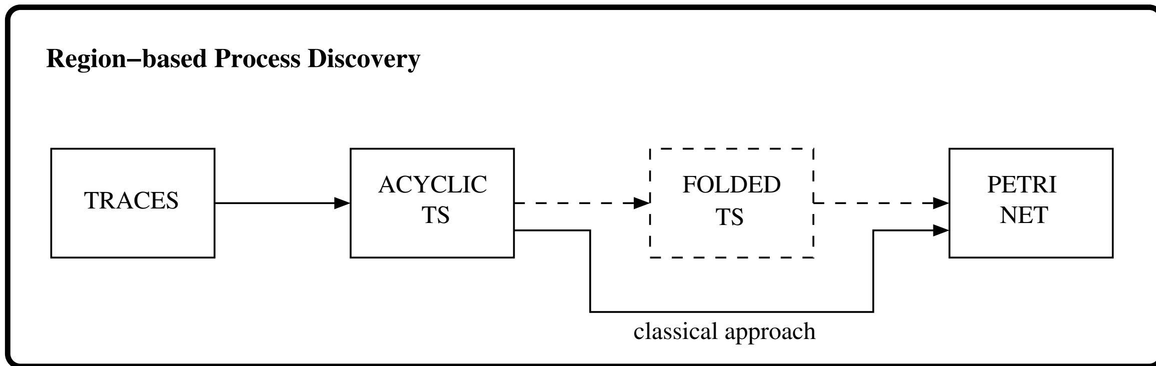



Fig. 1. Incorporating folding in region-based process discovery techniques.

Petri net that has a behavior *equivalent* to a given transition system. For this problem, several approaches in the literature have appeared to devise practical algorithms for synthesis. In [7] polynomial algorithms for the synthesis of bounded Petri nets were presented, translating the synthesis problem to the resolution of Integer Linear Programming (ILP) models. Recently, an extension of this approach has been presented in [8]. In [9], the theory of regions was applied for the synthesis of safe Petri nets with *bisimilar* behavior. This theory has been extended to bounded Petri nets [10].

Process Discovery differs from synthesis in the knowledge assumption: while in synthesis one assumes a complete description of the system, only a partial description of the system is assumed in Process Discovery. Therefore, equivalence or bisimulation is not a goal to achieve in Process Discovery. Instead, obtaining approximations that succinctly represent the log under consideration are more valuable [11]. However, synthesis can be adapted for Process Discovery in two ways: either the log is encoded as a transition system (introducing state information, as described in [12]) and state-based methods are applied [13], or language-based methods are used directly on the log [14], [15]. In this paper we follow the first approach.

The transition system that is obtained from a log is acyclic [11]. Then, if iterative behavior exists (and this happens if the process to reveal is non-trivial), the corresponding acyclic transition system must contain unfolded behavior of such iterations by means of repetitive patterns. This implies that the size of this transition system is typically large, and therefore, given that the complexity of region-based techniques depend on the size of the input transition system, these techniques suffer when handling this type of transition systems.

In this paper we propose and intermediate step to the applied in region-based Process Discovery (see Fig. 1). A folding step is performed before of applying a process discovery technique, which allows to factor out the cycles and therefore reducing the size of the transition system. In contrast to approaches in the literature to detect cycles [16], [17], [18], [19], [20], [21], the technique presented in this paper applies the ideas presented in a recent work that uses a basis of state regions for process discovery [22].

In summary, the main contributions of this paper are:

- *Folding strategies for transition systems obtained from logs* that, when discovering $k$-bounded nets, allow reducing the total number of states and speed-up the discovery process.
- Detecting folding opportunities can be computationally expensive, Thus *algorithms and data structures* specially tailored to efficiently solve this problem are presented.
- The theory of this paper has been implemented in a tool.The experimental results reported demonstrate the significance and effectiveness of this theory.

### *A. Organization*

We start by giving the necessary background in Sect. II. Section Sect. III describes the conversion of a log into a transition system and techniques for its later reduction. Section Sect. IV contains the main

algorithms and data strutures to find repetitive behavior by applying the theory of regions. The techniques of this paper are evaluated in Sect. V.

## II. Background

### A. Finite Transition Systems and Petri Nets

*Definition 1 (Transition system):* A *transition system* (TS) is a tuple $\langle S, \Sigma, T, \mathsf{s}_0 \rangle$, where $S$ is a set of *states*, $\Sigma$ is an alphabet of *actions*, $T \subseteq S \times \Sigma \times S$ is a set of *(labelled) transitions*, and $\mathsf{s}_0 \in S$ is the *initial state*.

We use $\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}'$ as a shortcut for $(\mathsf{s}, \mathsf{e}, \mathsf{s}') \in T$, and we denote its transitive closure as $\xrightarrow{*}$. A state $\mathsf{s}'$ is said to be *reachable from state* $\mathsf{s}$ if $\mathsf{s} \xrightarrow{*} \mathsf{s}'$. We extend the notation to transition sequences, i.e., $\mathsf{s}_1 \xrightarrow{\sigma} \mathsf{s}_{n+1}$ if $\sigma = \mathsf{e}_1 \ldots \mathsf{e}_n$ and $(\mathsf{s}_i, \mathsf{e}_i, \mathsf{s}_{i+1}) \in T$. Similarly, we use $\mathsf{s} \overset{\mathsf{e}}{\text{---}} \mathsf{s}'$ as a shortcut for $\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}' \vee \mathsf{s}' \xrightarrow{\mathsf{e}} \mathsf{s}$, being $\overset{*}{\text{---}}$ its transitive closure. A state $\mathsf{s}'$ is said to be *connected to state* $\mathsf{s}$ if $\mathsf{s} \overset{*}{\text{---}} \mathsf{s}'$. Let $A = \langle S, \Sigma, T, \mathsf{s}_0 \rangle$ be a TS. We consider TSs in which all states are connected and satisfy the following axioms: i) $S$ and $\Sigma$ are finite sets, ii) every event has an occurrence and iii) every state is reachable from the initial state. The *language* of a TS $A$, $\mathcal{L}(A)$, is the set of traces feasible from the initial state.

Two TSs can be compared using the following relation:

*Definition 2 (Simulation, Bisimulation [23]):* Let $A = (S, \Sigma, T, \mathsf{s}_0)$ and $A' = (S', \Sigma, T', \mathsf{s}'_0)$ be two TSs with the same set of events. A *simulation* of $A$ by $A'$ is a relation $\pi$ between $S$ and $S'$ such that

- for every $\mathsf{s}_i \in S$, there exists $\mathsf{s}'_i \in S'$ such that $\mathsf{s}_i \pi \mathsf{s}'_i$.
- for every $\mathsf{s}_i \xrightarrow{\mathsf{e}} \mathsf{s}_j \in T$ and for every $\mathsf{s}'_i \in S'$ such that $\mathsf{s}_i \pi \mathsf{s}'_i$, there exists $\mathsf{s}'_i \xrightarrow{\mathsf{e}} \mathsf{s}'_j \in T'$ such that $\mathsf{s}_j \pi \mathsf{s}'_j$.

When $A$ is simulated by $A'$ with relation $\pi$, and viceversa with relation $\pi^{-1}$, $A$ and $A'$ are *bisimilar* [23] and $\mathcal{L}(A) = \mathcal{L}(A')$.

*Definition 3 (Petri net [24], pure Petri net):* A Petri net (PN) is a tuple $(P, T, W, M_0)$, where $P$ and $T$ represent finite sets of places and transitions, respectively, and $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the weighted flow relation. The initial marking $M_0 \in \mathbb{N}^{|P|}$ defines the initial state of the system. When the flow relation satisfies that, for all transition $t$ and place $p$, $W(t,p) \cdot W(p,t) = 0$, then the Petri net is *pure*.

The sets of input and output transitions of place $p$ in PN $N$ are denoted by ${}^\bullet p$ and $p^\bullet$, respectively. A transition $t \in T$ is *enabled* in a marking $M$ if $\forall p \in P : M(p) \geq W(p,t)$. Firing an enabled transition $t$ in a marking $M$ leads to the marking $M'$ defined by $M'(p) = M(p) - W(p,t) + W(t,p)$, for $p \in P$, and is denoted by $M \xrightarrow{t} M'$. The set of all markings reachable from the initial marking $M_0$ is called its Reachability Set. If for any place $p$ and reachable marking $M$, it holds that $M[p] \leq k$, then the net is said to be *$k$-bounded*. The *Reachability Graph* of $N$, denoted $\mathsf{RG}(N)$, is a transition system in which the set of states is the Reachability Set, the events are the transitions of the net and a transition $(M_1, t, M_2)$ exists if and only if $M_1 \xrightarrow{t} M_2$. We use $\mathcal{L}(N)$ as a shortcut for $\mathcal{L}(\mathsf{RG}(N))$.

For instance Fig. 3(b) shows the reachability graph of the PN in Fig. 3(a).

### B. Generalized Regions

The theory of regions [6], [25] provides a way to derive a Petri net from a transition system. Intuitively, a region corresponds to a place in the derived Petri net. In the initial definition, a region was defined as a subset of states of the transition system satisfying an homogeneous relation with respect to the set of events. Later extensions [26], [27], [10] generalize this definition to multisets, which is the notion used in this paper.

*Definition 4 (Multiset, $k$-bounded Multiset, Subset):* Given a set $S$, a multiset $r$ of $S$ is a mapping $r : S \to \mathbb{Z}$. The number $r(s)$ is called the *multiplicity* of $s$ in $r$. Multiset $r$ is $k$-bounded if all its multiplicities are less or equal than $k$. Multiset $r_1$ is a *subset* of $r_2$ ($r_1 \subseteq r_2$) if $\forall s \in S \ : \ r_1(s) \leq r_2(s)$.

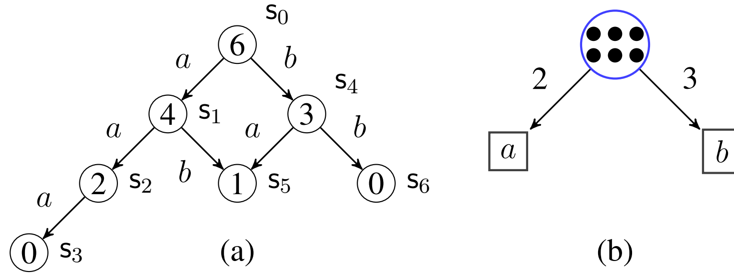



Fig. 2. (a) Region in a $\mathsf{TS}$: $r(\mathsf{s}_0) = 6, r(\mathsf{s}_1) = 4, \ldots, r(\mathsf{s}_6) = 0$, (b) corresponding place in the Petri net.

We define the following operations on multisets: $\begin{cases} \textit{Maximum power} & \mathrm{pow}(r) = \max_{s \in S} r(s) \\ \textit{Minimum power} & \mathrm{minp}(r) = \min_{s \in S} r(s) \\ \textit{Scalar sum}(k \in \mathbb{Z}) & (r+k)(s) = r(s) + k \\ \textit{Sum} & (r_1 + r_2)(s) = r_1(s) + r_2(s) \end{cases}$

Given a multiset $r$ defined over the set of states of a $\mathsf{TS}$ $A$, and a transition $\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}'$ in $A$, the *gradient of the transition* is defined as $\delta_r(\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}') = r(\mathsf{s}') - r(\mathsf{s})$. If all the transitions of an event $\mathsf{e}$ have the same gradient, we say that the event $\mathsf{e}$ has *constant gradient*, whose value is denoted as $\delta_r(\mathsf{e})$.

*Definition 5 (Region):* A *region* $r$ is a multiset defined in a $\mathsf{TS}$, in which all the events have constant gradient.

*Example 1:* Fig. 2(a) shows a $\mathsf{TS}$. The numbers within the states correspond to the multiplicity of the multiset $r$ shown. Multiset $r$ is a region because both events $\mathsf{a}$ and $\mathsf{b}$ have constant gradient, *i.e.* $\delta_r(\mathsf{a}) = -2$ and $\delta_r(\mathsf{b}) = -3$. There is a direct correspondence between regions and places of a $\mathsf{PN}$. The gradient of the region defines the flow relation of the corresponding place, and the multiplicity of the initial state indicates the number of initial tokens [10]. Fig. 2(b) shows the place corresponding to the region shown in Fig. 2(a).

We say that region $r$ is *normalized* if $\mathrm{minp}(r) = 0$. Any region $r$ can become normalized by subtracting $\mathrm{minp}(r)$ from the multiplicity of all the states. We denote by $\downarrow\! r$ the normalization of a region $r$, so that $\downarrow\! r = r - \mathrm{minp}(r)$.

It is useful to define a normalized version of the sum operation between regions, since it is closed in the class of normalized regions.

*Definition 6 (Normalized sum):* Let $r_1$ and $r_2$ be normalized regions, we denote by $r_1 \oplus r_2$ their *normalized sum*, so that $r_1 \oplus r_2 = \downarrow\!(r_1 + r_2)$.

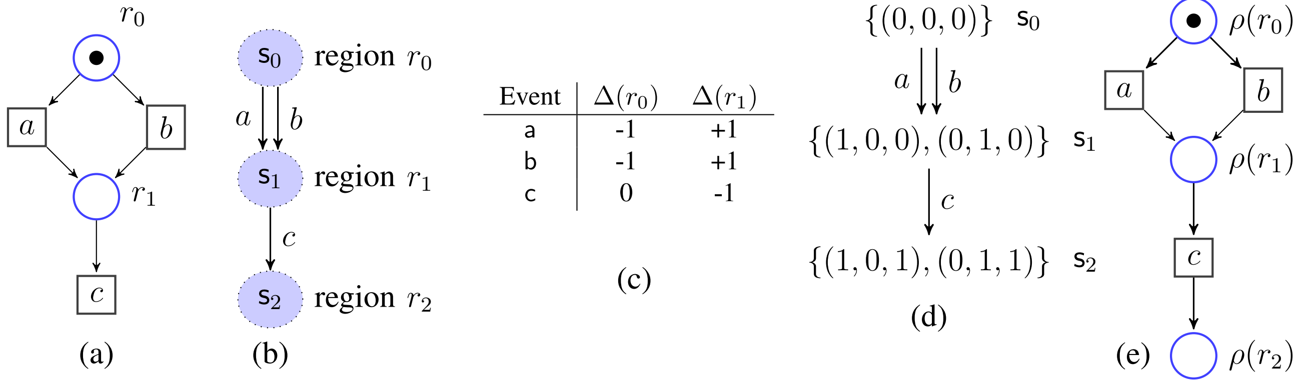
*Definition 7 (Gradient vector):* Let $r$ be a region of a $\mathsf{TS}$ whose set of events is $\Sigma = \{\mathsf{e}_1, \mathsf{e}_2, \ldots, \mathsf{e}_n\}$. The *gradient vector of* $r$, denoted as $\Delta(r)$, is the vector of the event gradients, i.e. $\Delta(r) = (\delta_r(\mathsf{e}_1), \delta_r(\mathsf{e}_2), \ldots, \delta_r(\mathsf{e}_n))$. Regions can be partitioned into classes using their gradient vectors.

*Definition 8 (Canonical region):* Two regions $r_1$ and $r_2$ are said to be equivalent if their gradient is the same, *i.e.* $r_1 \equiv r_2 \Leftrightarrow \Delta(r_1) = \Delta(r_2)$. Given a region $r$, the equivalence class of $r$, is defined as $[r] = \{r_i |\ r_i \equiv r\}$. A *canonical region* is the normalized region of an equivalence class, *i.e.* $\downarrow\! r$.

An example of canonical region is provided in Fig. 3(b), where a $\mathsf{TS}$ is shown in which some regions have been shadowed. The canonical region $r_1 = \{\mathsf{s}_1\}$ has gradient vector $\Delta(r_1) = (+1, +1, -1)$, using the event order $(\mathsf{a}, \mathsf{b}, \mathsf{c})$.

*Definition 9 (Subregion, Empty region, Minimal canonical region):* $r_1$ is a *subregion* of $r_2$, denoted as $r_1 \sqsubseteq r_2$, if $\downarrow\! r_1 \subseteq \downarrow\! r_2$. We denote by $\emptyset$ the region in which all states have zero multiplicity. A *minimal canonical region* $r$ satisfies that for any other region $r'$, if $r' \sqsubseteq r$ then $r' \equiv \emptyset$.

A $\mathsf{PN}$ built from the set of minimal canonical regions has the same language as a $\mathsf{PN}$ built using all the regions [25], thus it yields the smallest overapproximation with respect to the language of the $\mathsf{TS}$ [10]. That is, there is no other $\mathsf{PN}$ that includes the language of the $\mathsf{TS}$ and has a smaller language.




| Event | $\Delta(r_0)$ | $\Delta(r_1)$ |
|---|---|---|
| a | -1 | +1 |
| b | -1 | +1 |
| c | 0 | -1 |

Fig. 3. (a) A PN. (b) Its reachability graph, in which three regions, that form the set of minimal canonical regions, have been shadowed. Only two of them are required to obtain a basis. (c) Gradients of the regions in one of the possible bases. (d) TS with Parikh vector conflicts. Each state has been annotated with its set of Parikh vectors (event order $(\mathsf{a}, \mathsf{b}, \mathsf{c})$). (e) A PN with one redundant place.

### C. Gradient basis and region basis

The set of canonical regions of a TS, together with the normalized sum operation ($\oplus$), forms a free Abelian group [27]. Consequently, there exists a *basis* (*i.e.* subset of the group) such that every element in the group can be rewritten as a unique linear combination of the basis elements [28]. In particular all the minimal canonical regions can be generated from the basis [22].

*Example 2:* In TS of Fig. 3(b), the set of minimal canonical regions is formed by $r_0 = \{\mathsf{s}_0\}$, $r_1 = \{\mathsf{s}_1\}$ and $r_2 = \{\mathsf{s}_2\}$. However, we can express $r_2$ in terms of the other regions as $r_2 = -r_0 - r_1$. Note that, without normalizing the resulting regions it might be difficult to see the equivalence. For instance $-r_0 - r_1 = \{-\mathsf{s}_0, -\mathsf{s}_1\}$ which requires to subtract $-1$ (add 1) to each state multiplicity to obtain a normalized region, thus $\{-\mathsf{s}_0, -\mathsf{s}_1\} + 1 = \{\mathsf{s}_2\} = r_2$. Since $r_2$ is a linear combinations of the other regions, a possible basis is formed by only two regions, namely $r_0$ and $r_1$, whose gradient vectors appear in Fig. 3(c).

There exist efficient methods to find the region basis of a TS [22], based in the observation that any two paths between two states must have the same gradient in order to assign the same multiplicity to every state, no matter which path is taken to reach it. This requirement is formalized using the concepts of *Parikh vector* and *Parikh vector conflict*.

*Definition 10 (Parikh vector, Parikh vector conflict):* Given a TS $A = \langle S, \Sigma, T, \mathsf{s}_0 \rangle$, the *Parikh vector of a sequence* $\sigma$ is a vector $p_\sigma \in \mathbb{N}^{|\Sigma|}$ such that $p_\sigma(\mathsf{e}) = \#(\sigma, e)$, where $\#(\sigma, \mathsf{e})$ denotes the number of times that event $\mathsf{e}$ occurs in $\sigma$. The *set of Parikh vectors of a state* $\mathsf{s}$, denoted as $P_\mathsf{s}$, contains the Parikh vectors of all sequences $\sigma$ that start from $\mathsf{s}_0$ and end in $\mathsf{s}$. If some state $\mathsf{s}$ has $|P_\mathsf{s}| > 1$, we say that the state (and the whole TS) has a *Parikh vector conflict*.

For instance consider the TS of Fig. 3(b). The same TS but annotated with the Parikh vectors of each state is shown in Fig. 3(d).The TS has a Parikh vector conflict since it contains two states with more than one Parikh vector assigned. This imposes some restrictions on the gradients that correspond to a region. For instance, gradient $(+1, 0, 0)$ (with event order $(\mathsf{a}, \mathsf{b}, \mathsf{c})$) actually does not correspond to a region, since state $\mathsf{s}_1$ can be reached from $\mathsf{s}_0$ by a path in which event $\mathsf{a}$ occurs or by a path in which event $\mathsf{b}$ occurs: if we assume that the multiplicity assigned to $\mathsf{s}_0$ is 0, then in the first case the multiplicity assigned to $\mathsf{s}_1$ should be 1 as the gradient of $\mathsf{a}$ is $+1$, but in the second case should be also 0 because the gradient of $\mathsf{b}$ is 0.

The same reasoning can be made by looking at the Parikh vector conflicts. The difference between any pair of Parikh vectors assigned to a given state reveal combinations of event gradients that must be zero in order to have a consistent assignation of multiplicities to states. For instance, consider the Parikh vector conflict in $\mathsf{s}_1$. The difference between the two vectors is $(1, 0, 0) - (0, 1, 0) = (1, -1, 0)$. Thus, any region $r$ must satisfy $(1, -1, 0) \cdot \Delta(r)^T = 0$, that is $\delta_r(\mathsf{a}) - \delta_r(\mathsf{b}) = 0$, which can be rewritten as $\delta_r(\mathsf{a}) = \delta_r(\mathsf{b})$.

Since the presence of conflicts increases the number of restrictions on the gradients of regions, the size of the basis is inversely related to the number of linearly independent conflicts. Note that if the TS has no conflicts, then the size of the basis is the number of events in the TS [27][1]. In such situation the *standard basis* can be used, which is the region basis whose gradient basis is composed by all linearly independent unity vectors in which only one event gradient is different than zero. For instance the TS of Fig. 2(a) has no conflict, thus its gradient basis has size two (since it has only two different events), and we could use the standard gradient basis $\{(1,0),(0,1)\}$ with event order (a, b).

### D. Region-based process discovery algorithms

In this paper we focus on region-based algorithms for Process Discovery.

*Definition 11 (Region-based discovery algorithm, $k$-bounded discovery):* A *region-based* discovery algorithm $\mathcal{M}$ is any algorithm that, given a TS $A$, builds a PN $\mathcal{M}(A)$ by computing a finite set of regions in the TS such that no other PN built from the regions of the TS has a smaller language. Moreover, the algorithm can be required to produce a $k$-bounded output PN, denoted $\mathcal{M}_k(A)$, in which case the search is limited to the $k$-bounded regions of the TS. In this latter case we say that the algorithm performs the *$k$-bounded discovery* of $A$.

### E. Comparing regions between TSs

In forthcoming sections it will be important to compare two TSs with different sets of states but with the same alphabet of events, containing analogous regions. For instance, consider the TSs (a) and (b) in Fig. 4. Both TSs have the same language and every region in one of the TSs has an equivalent region with the same gradient and multiplicity in the initial state in the other TS, although the set of regions is different because the states in which they are defined are different.

To ease the comparison between regions of different TSs, we use an alternative representation of a region that abstracts states. As it is possible to determine the multiplicity of any state in a TS given the gradient of the region and the multiplicity of one of the states [22], we use this information as an alternative way of representing a region. Since this way of representing a region is directly related to the corresponding place of the region in a PN, we use the term *place* also to refer to a region in this alternative representation.

*Definition 12 (place of a region, places of a TS):* A *place* of a region $r$ in TS $A = \langle S, \Sigma, T, \mathsf{s}_0\rangle$, denoted $\rho(r)$, is the tuple $\langle {\downarrow} r(\mathsf{s}_0), \Delta(r)\rangle$. The set of places of $A$, denoted as $\Pi_A$, is the set of places of all the regions in $A$, *i.e.* $\Pi_A = \{\rho(r) \mid \text{r is a region in } A\}$. Similarly the set of $k$-bounded places, denoted $\Pi^k_A$ corresponds to the places of the set of $k$-bounded regions of $A$.

Given a TS $A = \langle S, \Sigma, T, \mathsf{s}_0\rangle$, the pure PN $N$ with the smallest language such that $\mathcal{L}(A) \subseteq \mathcal{L}(N)$ can be built from $\Pi_A$ by:

- Adding one place $p$ to $N$ for each place $\rho(r) = \langle {\downarrow} r(\mathsf{s}_0), \Delta(r)\rangle$ in $\Pi_A$, with an initial number of tokens equal to ${\downarrow} r(\mathsf{s}_0)$, and
- Defining the flow relation $W$ of $N$ for place $p$ as $\begin{cases} W(t,p) = \delta_r(t), W(p,t) = 0 & \text{if } \delta_r(t) > 0 \\ W(t,p) = 0, W(p,t) = -\delta_r(t) & \text{otherwise.}\end{cases}$

However, usually not all places restrict the language of the net. A place $\rho(r)$ is said to be *redundant* with respect to a set of places $\Pi$ if a PN built from $\Pi$ has the same language as a PN built from $\Pi - \{\rho(r)\}$.

*Example 3:* Consider the TS of Fig. 3(b). Let $\Pi$ the set of places of the minimal canonical regions, *i.e.* $\Pi = \{\rho(r_0), \rho(r_1), \rho(r_2)\}$. The PN built from $\Pi$ is shown in Fig. 3(e). This PN contains one redundant place, $\rho(r_2)$, that can be removed without changing the language of the net, yielding the PN in Fig. 3(a).

[1]If the number of events is smaller than the number of states, which is the usual case.

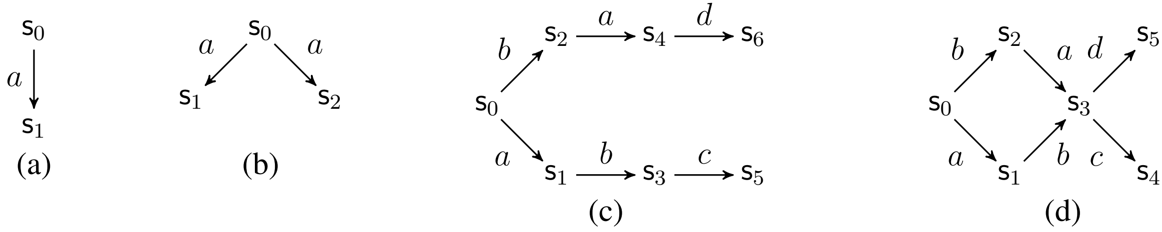



Fig. 4. (a) A deterministic TS. (b) Non-deterministic equivalent TS. The languages of both TSs is the same and every region in one of the TSs has an equivalent region with the same gradient and number of tokens in the other TS. (c) A TS obtained by the sequential conversion of the log $\{abc, bad\}$. (d) Its quotient TS (the multiset conversion of the log as well). The quotient TS accepts more sequences, but their PN is the same.

## III. Converting a log into a transition system

The classical theory of regions is defined on TSs, while the typical input in Process Mining is a set of sequences. To be able to use the theory of regions in such scenario, two solutions have been proposed. The first one is to adapt the theory of regions to languages (language mining) [14], [15], and the other is to convert the language into a TS and then apply the classical theory of regions. In this section we review the state of the art of this latter option, since language-based regions theory can be seen as the classical region theory using a particular type of conversion [29].

In [30] three conversions from a language to a TS were proposed, namely *sequence*, *multiset*and *set*. The main difference between the conversions is how it is decided whether the occurrence of an event in a trace produces a new state in the TS or just introduces an arc to an existing state. In this paper we will focus on the first two conversions, since the *set* conversion is not guaranteed to preserve the same set of regions as the other two do.

In the *sequence* conversion, two sequences lead to the same state if the order of the events in both sequences is the same. For instance if $L = \{abc, bad\}$, the TS obtained from this conversion is shown in Fig. 4(c). In the *multiset* conversion different event orders are allowed, but still it is required that the number of occurrences of each event to be equal. With the previous log, this yields the TS of Fig. 4(d). Let us formalize these two conversions:

*Definition 13 (Sequential, multiset conversion):* Given a log $L$, we say that a sequence $w$ is a *prefix* of $L$ if it exists some (possibly empty) sequence $\sigma$ such that, $w\sigma \in L$. The *sequential* conversion of $L$, is a TS $\langle S, \Sigma, T, \mathsf{s}_\epsilon \rangle$, denoted as $\mathsf{TS}_{\text{seq}}(L)$, such that: $S$ contains one state $\mathsf{s}_w$ for each prefix $w$ in $L$, with $\epsilon$ denoting the empty prefix, and $T = \{\mathsf{s}_w \xrightarrow{\mathsf{e}} \mathsf{s}_{w\mathsf{e}} \mid w\mathsf{e} \text{ is a prefix of } L\}$.

The *multiset* conversion of $L$, denoted as $\mathsf{TS}_{\text{mset}}(L)$, is a TS $\langle S, \Sigma, T, \mathsf{s}_{p_\epsilon} \rangle$, such that: $S$ contains one state $\mathsf{s}_{p_w}$ for each Parikh vector $p_w$ of a prefix $w$ in $L$, and $T = \{\mathsf{s}_{p_w} \xrightarrow{\mathsf{e}} \mathsf{s}_{p_{w\mathsf{e}}} \mid w\mathsf{e} \text{ is a prefix of } L\}$.

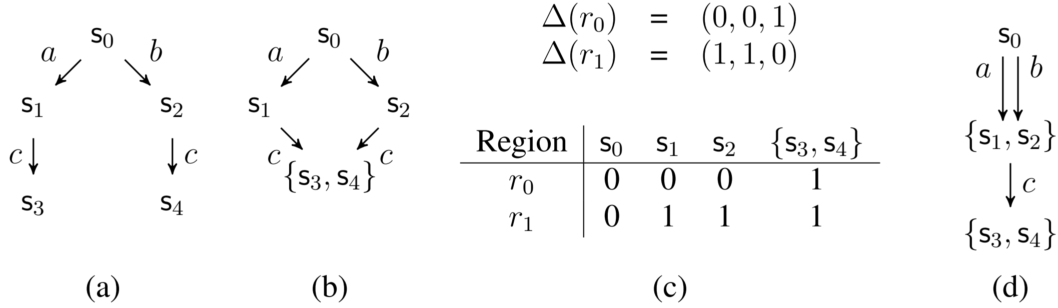
It was proved in [22] that both conversions yield TSs with the same set of places.

*Property 1:* Given a log $L$, $\Pi_{\mathsf{TS}_{\text{seq}}(L)} = \Pi_{\mathsf{TS}_{\text{mset}}(L)}$.
This is important because it proves that both conversions provide the same information to a region-based algorithm and the multiset conversion usually produces smaller TSs, thus potentially affecting the performance of the algorithms.

### A. Reduction techniques for TSs

In [22] an aggressive reduction technique was proposed, that considerably diminishes the size of the TS at the cost of forbidding some specific regions. The technique, named *common final marking* (CFM) reduction, has two steps:

- From a TS $A$, create a TS $A'$ by merging all *sink states* (states without outgoing arcs) into a single state. We say that $A'$ is the *single sink version of* $A$.
- Build the *quotient TS* by identifying all the states that have the same multiplicity in all the regions, thus are indistinguishable, and then merging them.




| Region | $s_0$ | $s_1$ | $s_2$ | $\{s_3, s_4\}$ |
|---|---|---|---|---|
| $r_0$ | 0 | 0 | 0 | 1 |
| $r_1$ | 0 | 1 | 1 | 1 |

Fig. 5. (a) A TS. (b) Its single sink version. (c) Gradient and region basis of the single sink version. States $s_1$ and $s_2$ have the same multiplicity in any of the regions in the basis, so they are equivalent and can be merged. (d) Corresponding quotient TS.

The basic idea is that the sink states of a TS they all correspond to the same final state, a reasonable assumption in process mining, so they can be merged in a unique state, yielding a bisimilar TS. This enforces that the final marking of the net, in all cases, to be the same. Once this has been done, equivalent states are identified and merged.

*Definition 14:* States $s$ and $s'$ in a TS are said to be equivalent, $s \equiv s'$, if, for all region $r$ of the TS, $r(s) = r(s')$. We denote the equivalence class of state $s$ as $[s]$.

Identifying equivalent states can be reduced to checking the multiplicities in the basis of regions (see Sect. II-C), as the following proposition states.

*Proposition 1 ([28]):* States $s$ and $s'$ in a TS $A$ are equivalent if, for all region $r$ in the region basis of $A$, $r(s) = r(s')$.

The state equivalence relation partitions the set of states into equivalence classes. The TS that abstracts the behavior of a given TS at the level of the equivalence classes is called the *quotient TS* .

*Definition 15 (Quotient TS):* Let $A = \langle S, \Sigma, T, s_0 \rangle$ be a TS. The *quotient TS of* $A$, denoted $A_{/\equiv}$, is a TS $\langle S_{/\equiv}, \Sigma, T_{/\equiv}, [s_0] \rangle$, where $S_{/\equiv} = \{[s] \mid s \in S\}$ and $T_{/\equiv} = \{[s] \xrightarrow{e} [s'] \mid s \xrightarrow{e} s' \in T\}$.

A fundamental result is that the quotient TS has the same set of places as the original TS.

*Theorem 1 ([22]):* Let $A$ be a TS, $A_{/\equiv}$ be its quotient TS and $\mathcal{M}$ be a region-based mining algorithm. Then, $\Pi_A = \Pi_{A_{/\equiv}}$ and $\mathcal{L}(A) \subseteq \mathcal{L}(A_{/\equiv}) \subseteq \mathcal{L}(\mathcal{M}(A))$.

For instance the set of places of the TS in Fig. 4(c) is the same as its quotient TS in (d), however, clearly the language of (d) is a proper superset of the language of (c), as sequences $abd$ and $bac$ are not possible in (c).

On the other hand, by merging all sink states, some regions might no longer be feasible, thus the language of the mined PN from the CFM reduction might be a superset of the language of the PN mined from the original transition system.

*Theorem 2 ([22]):* Let $A$ be a TS, $A'$ its CFM reduction and $\mathcal{M}$ a region-based algorithm. Then, $\Pi_A \supseteq \Pi_{A'}$ and $\mathcal{L}(\mathcal{M}(A)) \subseteq \mathcal{L}(\mathcal{M}(A'))$.

For instance in Fig. 5 we can see a TS that could be derived from $L = \{ac, bc\}$. Both the sequential and the multiset conversion yield the same TS, shown in (a). This TS has two sink states $s_3$ and $s_4$, which can be merged obtaining a TS, depicted in (b), with the same language. By computing its region basis $\{r_0, r_1\}$, shown in (c), additional state equivalencies are found between $s_1$ and $s_2$ as the columns for both states are equal, meaning their multiplicities in all the regions in the basis is the same, thus they are equivalent (Proposition 1). The corresponding CFM reduction is depicted in (d).

Finally, note that for cyclic TSs that do not have sink states, the CFM reduction simply computes the quotient TS and no region is removed.

## IV. TS FOLDING FOR $k$-BOUNDED DISCOVERY

A TS obtained from a log either by sequential or multiset conversion does not have cycles, although many fragments of its sequences might represent unfolded behavior of real cycles in the system. In this

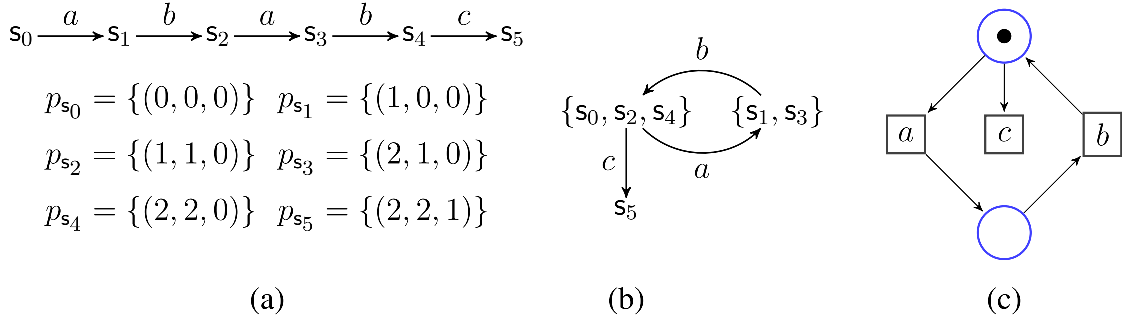



Fig. 6. (a) A TS together with the Parikh vectors assigned to each state (lexicographical event order). (b) Folded TS for 1-bounded (safe) discovery. (c) Mined PN in both cases.

section we will show how we can efficiently detect such unfoldings, to be able to fold them. The advantages of detecting cycles in a TS are two-fold: on the one hand new equivalent states are identified which reduces the size of the TS and, on the other hand, it introduces new Parikh vector conflicts that can reduce the size of the region basis. This is specially desirable since the performance of some discovery/synthesis algorithms depends crucially on the size of the region basis [28], [22], and a reduction on this size speeds-up the discovery process.

Now the question is how we can find unfolded cycles when performing a $k$-bounded discovery of a PN in a TS. First of all consider the following property:

*Property 2:* Given a TS $A$, if $A$ contains a cycle $c$, for any region $r$ of $A$, the sum of the gradients of $r$ for the events in the cycle $c$ must be zero. Formally $p_c \cdot \Delta(r)^T = 0$.
This is simply a particular case of Parikh vector conflict, since if this condition is not satisfied, then the multiplicities assigned to states are not stable as shown in Sect. II-C.

Thus, from the region point of view, a cycle is equivalent to the condition that a sum of gradients must be zero. This observation can be combined with the following lemma to find unfolded cycles in a TS.

*Lemma 1:* Given a TS $A$. For any region $r$ of $A$, if $A$ contains a sequence of events $\sigma = \alpha\beta^j\gamma$, with $j \geq k+1$, that is a sequence in which a subsequence $\beta$ is repeated at least $k+1$ times, then the sum of the gradients of $r$ for all the events in $\beta$ must be 0, or $\mathrm{pow}(r) \geq j \geq k+1$.

*Proof:* Consider $\sigma = \mathsf{s}_0 \xrightarrow{\alpha} \mathsf{s}_i \xrightarrow{\beta^j} \mathsf{s}_j \xrightarrow{\gamma}$, we will prove that it is not possible to have $p_\beta \cdot \Delta(r)^T \neq 0$ and $\mathrm{pow}(r) \leq k$. We know $r(\mathsf{s}_j) = r(\mathsf{s}_i) + j(p_\beta \cdot \Delta(r)^T)$, and that $\mathrm{pow}(r) \geq \|r(\mathsf{s}_j) - r(\mathsf{s}_i)\|$. Assume without loss of generality that $p_\beta \cdot \Delta(r)^T > 0$. The minimum value it can have is 1, as gradients are integers. Thus, the minimum value for $r(\mathsf{s}_j)$ is $r(\mathsf{s}_i) + j \cdot 1$, which implies that $\|r(\mathsf{s}_j) - r(\mathsf{s}_i)\| = j \geq k+1$, hence $\mathrm{pow}(r) \geq k+1$, contradicting $\mathrm{pow}(r) \leq k$. ■

This lemma shows that, if a given cycle is unfolded at least $k+1$ times, then this has the same effect as a cycle if we are discovering a $k$-bounded PN, since any $k$-bounded region of the TS must have the sum of the gradients of the events appearing in the repeated sequence equal to zero. Consequently, we can detect when a pattern is repeated at least $k+1$ times to identify cycles.

*Example 4:* Consider the TS of Fig. 6(a). Since it is acyclic and has no conflict (no state has more than one Parikh vector), we can use the standard gradient basis (of size three, since the system has three events) to mine it. On the other hand, the TS obtained by folding the repeated subsequence $\xrightarrow{a}\xrightarrow{b}$, shown in (b), contains one conflict, thus its gradient basis contains only two gradients. As an example of the achieved speed-up, the region-based algorithm implemented in the tool `rbminer` [31], which explores combinations of the regions in the basis, needs to explore 26 regions to achieve synthesis in the first case, and only 8 in the second case.

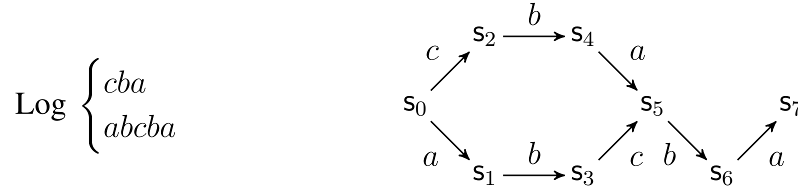



Fig. 7. (left) A log $L$ in which no trace contains a tandem repeat. (right) However, its corresponding transition system, $\mathsf{TS}_{\text{mset}}(L)$, contains tandem $baba$.

## A. Tandem repeats

The detection of unfolded cycles in an acyclic TS is a related problem to finding consecutively repeated patterns in a string. The latter problem has been studied in several fields with many variations and under different names [16], [17], [18], [19], [20], [21], although it is often referred as the *finding tandem repeats* problem. Formally, one of the most common formulations of the problem is, given a string $\omega$, find all substrings $\alpha$ such that $\omega = \beta\alpha^k\gamma$ for some $k \geq 2$. In the particular case of process discovery, it exists an implementation of one of these algorithms as a filter in the ProM tool [32], namely the *ExactTandemRepeat* filter, that simply removes all tandem repeats of all the sequences in the log.

Although the problem has been periodically revisited in the literature, up to our knowledge, none of these previous approaches has been specially devised to efficiently solve the problem when the input is not a single word, but a set of words succinctly represented by an acyclic TS. Applying the same algorithms directly to the log would not only be inefficient, but also can potentially reduce the number of cycles found, since an unfolding of that cycle might not appear until the TS is built from the log. For instance, consider the log in Fig. 7. None of its traces contains a tandem repeat, but the TS exhibits one, *i.e.* $(\mathsf{ba})^2$.

Consequently, to obtain all cycles with minimal modifications of the algorithms, all the words of the language of the TS should be checked, but this strategy is clearly still more inefficient. For instance, consider the log containing only two words on alphabet $\{\mathsf{a}, \mathsf{b}\}$, namely $(\mathsf{ab})^n$ and $(\mathsf{ba})^n$, for an arbitrary large natural $n$. The TS obtained by multiset conversion of this log, is a concatenation of $n$ concurrency diamonds, thus the language of the TS contains $2^n$ words.

The problem of finding all the tandem repeats can be solved by incrementally computing all the paths of a given length present in the TS. First of all, we start with length one, that is, all the paths in which only one event occurs, grouping the paths if the same event is fired. The beginning and ending states of each one of such paths is recorded, and consecutive repetitions are searched for.

Algorithms 1, 2 and 3 show the pseudo-code of the proposed algorithm. The `initialize_tandems` function simply returns the initialized the data structures with all the sequences of length one. In particular it returns a set of objects that contain the sequence of events considered ($p.seq$) together with an array of occurrences of such sequence ($p.occ[j].begin$ and $p.occ[j].end$ are the initial and final states of one of such occurrences). We assume that all states have a total order and the occurrences are sorted using the initial state as first key and the final state as second key. This allows a faster implementation of the detection of contiguous occurrences.

The `extend_tandem` function, precisely returns the number of times that a particular starting occurrence in an occurrence array has contiguous repetitions, together with the indexes of the contiguous occurrences in the array. It uses the `search` function that can take advantage of the fact that the array is sorted so that a binary search is possible. Note that it is not necessary to return a set of tuples, since all the conversions of a log into a TS always produce a deterministic system (otherwise some states can be trivially merged).

The `find_tandems` algorithm is the main algorithm. First of all, calls `initialize_tandems` to initialize the data structures. The set of patterns that have at least $k + 1$ repetitions is kept in variable $C$, which is initially the empty set. Actually the information stored is a tuple containing the initial state of the tandem and the sequence of events repeated.

**Algorithm 1** initialize_tandems

```
1:  function INITIALIZE_TANDEMS(⟨S, Σ, T, s_0⟩)
2:      P ← ∅
3:      for i ← 1 to |Σ| do
4:          p.seq = e_i
5:          j ← 1
6:          for all s --e_i--> s' ∈ T do
7:              p.occ[j].begin ← s
8:              p.occ[j].end ← s'
9:              j ← j + 1
10:         end for
11:         sort( p.occ )
12:         P ← P ∪ {p}
13:     end for
14:     return P
15: end function
16: function EXTEND_TANDEM(occ, j)
17:     ⟨repetition, F⟩ ← ⟨0, ∅⟩
18:     repeat
19:         ⟨found, j⟩ ← search( occ, bfs(occ[j].begin) + l, occ[j].end )
20:         F ← F ∪ {j}
21:         repetition ← repetition + 1
22:     until not found
23:     return ⟨repetition, F⟩
24: end function
```

Variable $l$ simply contains the current length of the patterns under consideration. The patterns of length $l$ are stored in $P$, from which the patterns of length $l+1$ are generated in variable $N$ using the `expand` function shown in Algorithm 3.

To alleviate the number of checks that must be performed to generate $N$, some additional information is used. First of all, it is obvious that to obtain a tandem repeat of at least $k+1$ repetitions, the number of occurrence of any pattern must be, at least $k+1$. A more elaborated heuristic is to use the *depth* of the initial states: the function `bfs` returns the bread first search number of a given state. This is the depth of the state, so that initial state $\mathsf{s}_0$ has depth 0, its immediate successors have depth 1 and so on. This assignment is unique since the TSs obtained from logs are acyclic (although variations that can handle cyclic TSs are not difficult to obtain, since basically we must use the smallest depth for any given state).

The depth of the states is used to prune patterns that will never have $k+1$ repetitions. Basically if the starting depth plus $k$ repetitions of the pattern are past the depth of the last occurrence, then it is impossible that we have $k+1$ contiguous repetitions.

Note that, once a $k$ tandem is found, any occurrence of the pattern creates a cycle, even if it is alone (not only tandems are folded). However this strategy might let some tandems undetected, since the merging of equivalent classes is not performed until all tandems have been found. We illustrate this phenomenon with an example.

*Example 5:* In Fig. 8(a) we can see a TS that contains two tandems of size two: namely, $ab$ and $abc$. However if we compute its 1-folding using the algorithm for tandem repeats, the TS of (b) is obtained, since when tandem $ab$ is found, no further expansion of it is performed, thus tandem $abc$ is never found.

A possible solution is to iteratively run the algorithm until no new tandem is found. However, we will see in the next section that a more powerful strategy is possible.

**Algorithm 2** find_tandems

```
1:  procedure FIND_TANDEMS(⟨S, Σ, T, s0⟩, k)
2:      P ← initialize_tandems( ⟨S, Σ, T, s0⟩ )
3:      ⟨C, l⟩ ← ⟨∅, 1⟩                                   ▷ Initialize pattern length to 1
4:      while |P| > 0 do
5:          N ← ∅
6:          for all p ∈ P do
7:              ⟨first, last⟩ ← ⟨bfs(p.occ[1].begin), bfs(p.occ[|p.occ|].begin)⟩
8:              Skip ← ∅
9:              if (|p.occ| > k) and (first + k · l ≤ last) then
10:                 for i ← 1 to |p.occ| do
11:                     if i ∉ Skip then
12:                         ⟨repetitions, F⟩ ← extend_tandem( p.occ, i )
13:                         if repetitions > k then
14:                             Skip ← Skip ∪ F                ▷ Skip occurrences already in tandem
15:                             C ← C ∪ {⟨p.occ[i].begin, p.seq⟩}
16:                         else
17:                             N ← expand( N, T, p )
18:                         end if
19:                     end if
20:                 end for
21:             end if
22:         end for
23:         ⟨P, l⟩ ← ⟨N, l + 1⟩
24:     end while
25: end procedure
```

**Algorithm 3** expand

```
1:  function EXPAND(N, T, p)
2:      for i ← 1 to |p.occ| do
3:          s ← p.occ[i].end
4:          for all s —e→ s′ ∈ T do
5:              if ∃p′ ∈ N : p′.seq = p.seq · e then
6:                  p′.occ[|p′.occ| + 1].begin ← p.occ[i].begin
7:                  p′.occ[|p′.occ| + 1].end ← s′
8:              else
9:                  p′.seq ← p.seq · e
10:                 p′.occ[1].begin ← p.occ[i].begin
11:                 p′.occ[1].end ← s′
12:                 N ← N ∪ {p′}
13:             end if
14:         end for
15:     end for
16:     return N
17: end function
```

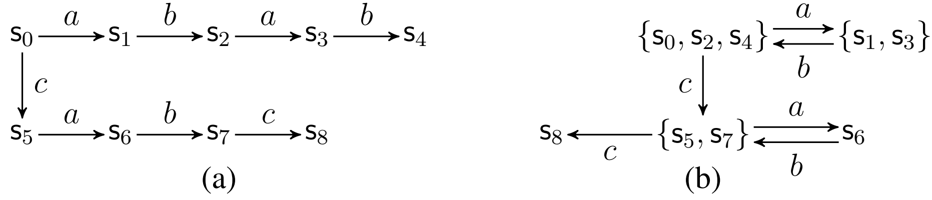


Fig. 8. (a) A TS containing two tandem repeats, but one of them "hidden" by the other. (b) Once the first tandem is folded, the second one is revealed.

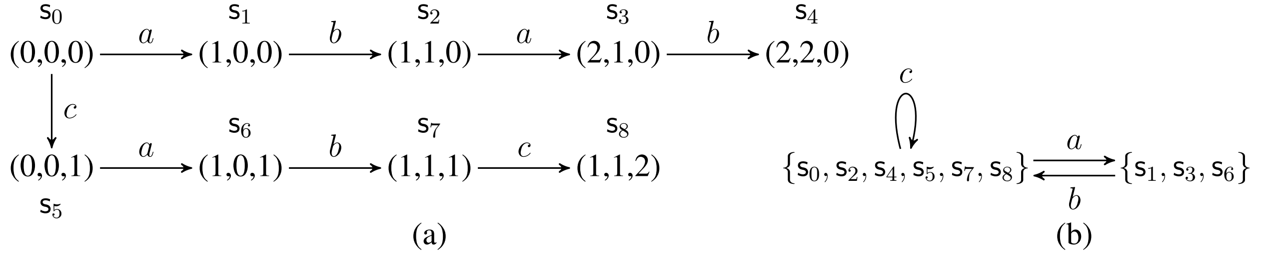


Fig. 9. (a) TS of Fig. 8(a) annotated with the Parikh vectors of each state. (b) TS after it has been fold by merging the equivalent states found with the Parikh trie.

### B. Beyond tandem repeats

While in previous subsection we have used the identification of exact tandem repeats as a mean to reduce the size of the TSs, in this section we do generalize this approach to tandems permutations. The reason is simple: if the gradient of a given pattern must be zero, then any permutation of the pattern will yield the same gradient. Consequently, instead of discovering exact tandems, we will use the Parikh vector of the states to find tandems permutations.

*Theorem 3:* Let $\mathsf{s}$ and $\mathsf{s}'$ be two states of a TS. In $k$-bounded discovery, if $p = p(\mathsf{s}) - p(\mathsf{s}')$ contains only multiples of a value $v > k$, then $\mathsf{s} \equiv \mathsf{s}'$.

*Proof:* Consider any region $r$ of the $k$-bounded PN. Since the net is $k$-bounded, $\mathrm{pow}(r) \leq k$. We will prove that in any of these regions, both states will have the same multiplicity. If $p \cdot \Delta^T(r) = 0$, then trivially $r(\mathsf{s}) = r(\mathsf{s}')$. If $p \cdot \Delta^T(r) \neq 0$, since all the value in $p$ are multiples of $v$, then $p \cdot \Delta^T(r)$ yields a multiple of $v$, say $q \cdot v$ with $q \neq 0$, consequently, $\mathrm{pow}(r) \geq q \cdot v > k$ and we have a contradiction. ■

Thus the problem reduces to detect pairs of states whose Parikh vector difference contains 0 or a multiple of a value greater than $k$. Once these differences have been identified, we can divide them by the greatest common divisor[2], and the obtained vector represents a cycle in the TS. Once all the differences are available, they form a nullspace basis from which a region basis can be computed, and then equivalent states can be easily detected using Proposition 1.

This approach is capable of discovering "hidden" tandems, as next example illustrates.

*Example 6:* In Fig. 8 there is the obvious tandem $abab$, represented by Parikh vector $(2, 2, 0)$ (see Fig. 9). Dividing by the greatest common divisor, we do obtain vector $(1, 1, 0)$, indicating that events a and b form a cycle. However, we have also $p(\mathsf{s}_2) = (1, 1, 0)$ and $p(\mathsf{s}_8) = (1, 1, 2)$. thus $p(\mathsf{s}_8) - p(\mathsf{s}_2) = (0, 0, 2)$. Dividing by the greatest common divisor, we do obtain vector $(0, 0, 1)$, showing that event c forms a cycle (*i.e.* a self-loop, thus its gradient must be zero).

To efficiently detect relevant Parikh vector differences, we use a trie to store all the Parikh vectors and a recursive algorithm working on top of that structure.

*Definition 16 (Parikh trie):* A *Parikh trie* of a TS $\langle S, \Sigma, T, \mathsf{s}_0 \rangle$ without Parikh vector conflicts is a rooted tree $\langle V, E, v_0 \rangle$ with $|\Sigma|$ levels, composed of the set of nodes $V$, the permutation $E$ of the events in $\Sigma$, that indicates which event is considered at each level of the tree, and the initial node $v_0$. Each node

[2]Remember that the greatest common divisor of a number $a$ and 0 is defined to be $a$.

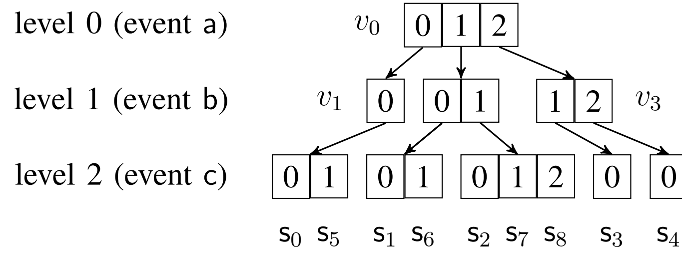



Fig. 10. Trie of Parikh vectors for TS in Fig. 8(a), with some of its nodes labeled.

$v \in V$ that is not a leaf is defined as a function $v : \mathbb{N} \to V$ that maps naturals to the set of nodes in the next level of $v$. If $v$ is a leaf, then $v : \mathbb{N} \to S$ that maps naturals to the set of states $S$.

The nodes are defined such that, for each state $\mathsf{s} \in S$, let $p_\mathsf{s}$ be its unique Parikh vector, and $v_0(p_\mathsf{s}(E_1))(p_\mathsf{s}(E_2)) \ldots (p_\mathsf{s}(E_{|\Sigma|})) = \mathsf{s}$, and no other path from root to leaf exists that does not represent the Parikh vector of a state in $S$.
The domain set of a function $v$ (thus of a node of a Parikh trie), that is the set of values in which it is defined, is denoted as $\text{dom}(v)$.

For instance, in Fig. 10 we can see the Parikh trie corresponding to the TS of Fig. 8(a). As an example, in this case $\text{dom}(v_0) = \{0, 1, 2\}$ and $\text{dom}(v_3) = \{1, 2\}$.

The algorithms to detect relevant Parikh vector differences are shown in Algorithms 4 and 5. For simplicity they assume that the trie contains more than one level (*i.e.* the root node is not a leaf).

**Algorithm 4** tandemSearch

```
 1: function TANDEM_SEARCH(⟨V, E, v0⟩, k, maxv)
 2:     maxv ← max_{v∈V} (dom(v))
 3:     Conflicts ← ∅
 4:     Used ← ∅
 5:     for i ← k + 1 to maxv do
 6:         if i ∉ Used then
 7:             m ← 0                                   ▷ Check all multiples of i, starting by 0
 8:             repeat
 9:                 for j ← 0 to max(dom(v0)) − m do
10:                     if {j, j + m} ⊆ dom(v0) then
11:                         tandem_search_recur( Conflicts, v0(j), v0(j + m), i )
12:                     end if
13:                 end for
14:                 Used ← Used ∪ {m}                   ▷ Mark multiple as already used
15:                 m ← m + i                           ▷ Next multiple of i
16:             until m > maxv
17:         end if
18:     end for
19:     return Conflicts
20: end function
```

The algorithms work as follows. First of all, the maximum number appearing in any Parikh vector is computed (line 2) and stored in $maxv$. This number is needed to know which is the maximum possible difference we must look for when comparing the Parikh vectors in the trie. Sets *Conflicts* and *Used* are initialized to the empty set. The former will contain the set of Parikh vector differences that identify equivalent states, according to Theorem 3 and the given $k$.

On the other hand, set *Used* is used to avoid replication of work during the search. Since all differences that are multiples of $i$ will be found in the recursive function, it is only necessary to call it using prime numbers. The *Used* set keeps track of which prime numbers have not been already used.

The algorithm first tries to find differences that are multiples of $k+1$ up to $maxv$ which is the maximum possible relevant difference between any two Parikh vectors, since vector $(0, 0, \ldots, 0)$ is always present. Variable $i$ always contains the number whose multiples the algorithm is looking for. Note that $0$ is considered a multiple since relevant Parikh vector differences can contain zeroes. After the first level, the search is recursively done in Algorithm 5.

**Algorithm 5** tandemSearchRecursive

```
1:  procedure TANDEM_SEARCH_RECUR(Conflicts, v_l, v_r, i)
2:      if leaf(v_l) then                                        ▷ Nodes v_l and v_r are leaves
3:          for all x ∈ dom(v_l), y ∈ dom(v_r) such that y − x ≡ 0 (mod i) do
4:              Conflicts ← Conflicts ∪ {p_{v_r(y)} − p_{v_l(x)}}    ▷ Add equivalency to set
5:          end for
6:      else                                                     ▷ v_l and v_r are internal nodes
7:          for all x ∈ dom(v_l), y ∈ dom(v_r) such that y − x ≡ 0 (mod i) do
8:              tandem_search_recur( Conflicts, v_l(x), v_r(y), i )
9:          end for
10:     end if
11: end procedure
```

Algorithm 5 is quite straightforward: if any of the nodes is a leaf node (the auxiliary function `leaf` returns true if the node is a leaf node), then both of them are, and all states that satisfy that their Parikh vector difference at the current level is a multiple of $i$ are found, and the complete Parikh vector difference is added to the result set *Conflicts*. For internal nodes, the same principle applies but a recursive call to the corresponding descendant nodes is issued instead (line 8).

For the sake of clarity some performance improvements (like an improved handling of the case where $v_r$ and $v_l$ are actually the same node) have not been added to the algorithm, although it simply consists in enforcing that $y \geq x$ to avoid duplicating work.

*Example 7:* We show how the algorithm works using the Parikh trie of Fig. 10 to fold the $\mathsf{TS}$ of Fig. 8(a) with $k = 1$. The `tandem_search` algorithm first calls the recursive algorithm with the following parameters: $v_l = v_0(0), v_r = v_0(0)$ and $i = 2$. This yields another call with $v_l = v_1(0), v_r = v_1(0)$ and $i = 2$, which finally finds the trivial conflict of state $\mathsf{s}_0$ with itself $(p_{\mathsf{s}_0} - p_{\mathsf{s}_0}) = (0, 0, 0))$. This sequence is illustrated as the first column in Fig. 11, where the $v_l$ parameter is depicted with horizontal lines, and the $v_r$ node with vertical lines (and a grid if both nodes are the same one). After the same conflict is found for $\mathsf{s}_1$, the first set of recursive calls is finished and the `tandem_search` algorithm issues the next recursive call with $v_l = v_0(1), v_r = v_0(1)$ and $i = 2$, which finds a new conflict (shown in the central column of Fig. 11). The next main recursive call has parameters $v_l = v_0(2), v_r = v_0(2)$ and $i = 2$, but does not yield any new conflict. Finally, the last top-level call with parameters $v_l = v_0(0), v_r = v_0(2)$ and $i = 2$ finds the last conflict, shown in the rightmost column of the figure.

### C. Comparison with CFM reduction

In this section we compare the folding strategy with the CFM reduction (see Sect. III-A). Although at first sight they might seem as orthogonal techniques, we will show that, in fact, CFM reduction is capable of folding unfolded cycles in many circumstances.

Before proving the main result of this section, we need to introduce the concept of *Parikh cycle basis*.

*Definition 17:* A *Parikh cycle basis* of a $\mathsf{TS}$ is the smallest set of cycles, such that the Parikh vector of any other cycle in the $\mathsf{TS}$ can be expressed as a linear combination of the Parikh vectors of the cycles in the basis.

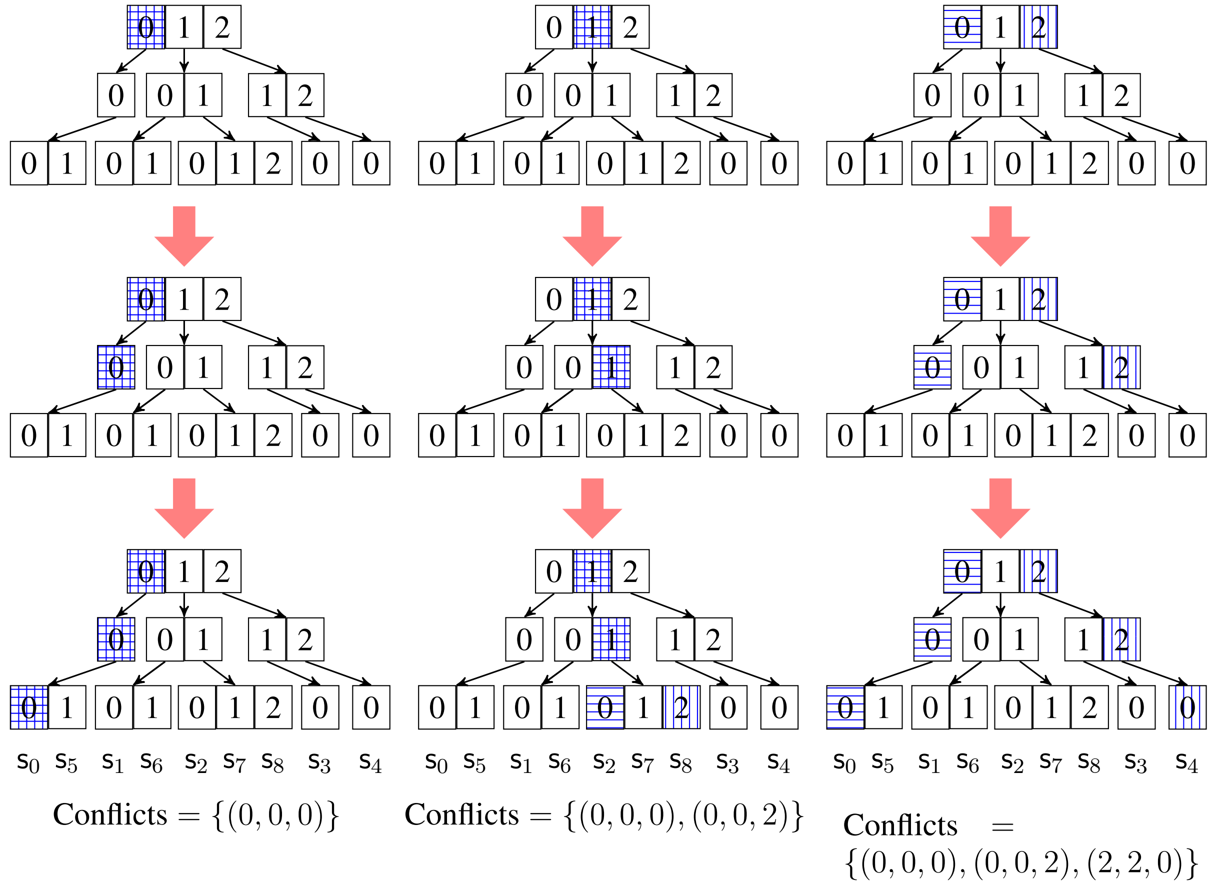



Fig. 11. Sequence of recursive calls performed by the Parikh trie folding algorithm for the TS of Fig. 8(a) and $k = 1$ (only the sequences yielding a new conflict are shown). The nodes visited as left node ($v_l$ in the algorithm) are depicted with horizontal blue lines. Similarly, vertical lines are used for $v_r$. If both nodes are the same, a grid is used instead.

A Parikh cycle basis is useful because if all the cycles in the basis satisfy a property and this property is not altered by linear combinations, then the property is shown to affect all cycles in the TS. This is the strategy that we follow in the next lemma.

*Lemma 2:* Let $N$ be a PN and $L \subseteq \mathcal{L}(N)$ be a log. If for every simple cycle in $\mathsf{RG}(N)$ (or every cycle in a Parikh cycle basis of $\mathsf{RG}(N)$) starting in some state s, the log contains a trace in which s is reached at least $i$ times with $i \geq 1$ with non-empty suffix $\sigma$, and contains another trace in which s is reached at least $i + 1$ times with the same suffix, then all cycles in $\mathsf{TS}_{\text{mset}}(L)$ will by folded by the CFM reduction.

*Proof:* $\mathsf{TS}_{\text{mset}}(L)$ is a bounded unfolding of $\mathsf{RG}(N)$. Consider the Parikh vectors of states in $\mathsf{TS}_{\text{mset}}(L)$ that map to s (*i.e.* the states s′ such that $f(\mathsf{s}') = \mathsf{s}$). The Parikh difference between these vectors will be a multiple of the cycle composition. Since both traces have the same suffix and sink states are merged, it appears a Parikh vector conflict equal to the Parikh difference that is a multiple of the cycle composition. Consequently, since it is required that all Parikh conflicts are valued to 0, then the cycle is enforced to have zero-greadient. As all the cycles in the Parikh cycle basis are zero-gradient, any cycle combination will also be, thus all cycles will effectively be folded when the quotient TS is built. ∎

The previous lemma has important consequences. First of all, it shows that in usual process discovery scenarios CFM reduction will yield smaller or equal TSs than the folding strategy. The following example illustrates this phenomenon.

*Example 8:* Fig. 12(a) shows a PN together with a log (b) that is a subset of the language of the net , from which several TSs can be obtained (c,d,e) depending on the type of conversion used. CFM reduction produces, as a side-effect, the folding of the unfolded cycles. However using the TS folding strategy we do not merge some of the sink states.

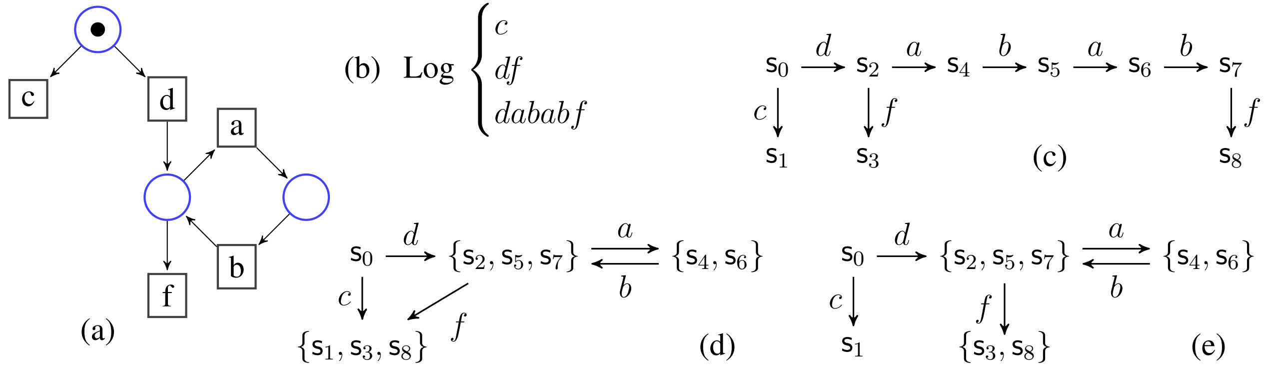



Fig. 12. (a) A PN. (b) A possible log. (c) TS obtained by multiset conversion. (d) TS after CFM reduction, it is isomorphic with the reachability graph of the PN. (e) TS obtained by folding unfolded cycles.

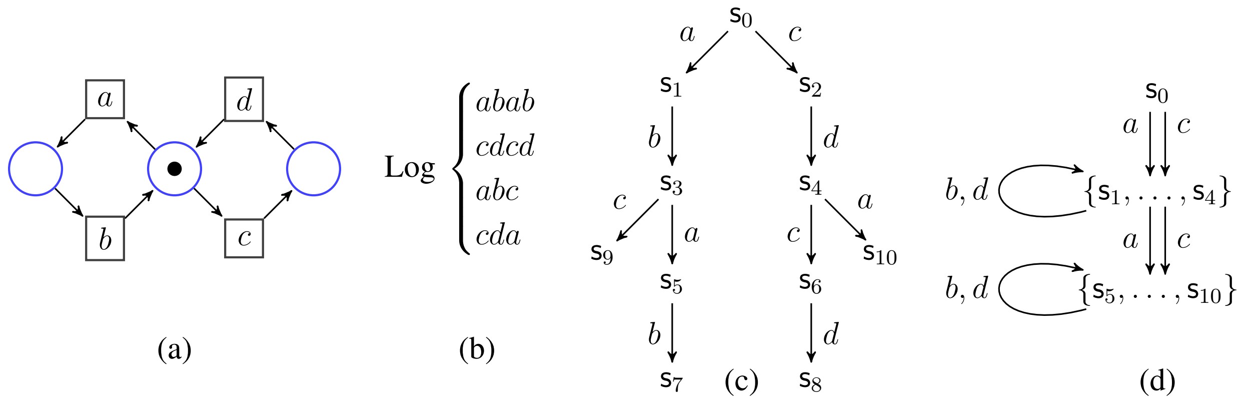



Fig. 13. (a) A 1-bounded PN. (b) A possible log of the PN. (c) TS obtained by multiset conversion. Mining this TS (using only 1-bounded regions) we would rediscover the PN. (d) TS after CFM reduction. It contains no 1-bounded region.

Note that CFM reduction does not always necessarily produce a smaller TS than using the folding strategy if the conditions of Lemma 2 are not fulfilled. For instance the following TS $s_0 \xrightarrow{a} s_1 \xrightarrow{b} s_2 \xrightarrow{a} s_3 \xrightarrow{b} s_4$ is not reduced by the CFM technique, but can be folded for 1-bounded discovery. The effectiveness of the CFM reduction is specially high in the presence of a special event used to indicate the completion of a case, a typical feature in process discovery logs [15].

However, Lemma 2 also shows that CFM reduction will always fold unfolded cycles, no matter the length of the cycle, thus it is not suitable for $k$-bounded discovery if $k \geq 2$. Moreover, CFM reduction can produce very rough overapproximations if the log contains many non-complete traces that end in different points of a cycle.

*Example 9:* In Fig. 13(a) we can see a safe PN. The log in (b) yields the TS of (c), that has the same set of 1-bounded places as the reachability graph of the PN. Thus, any region-based algorithm that mines (using $k = 1$) the $\mathsf{TS}_{\text{mset}}$ of the log, will return the original PN (or a PN with identical language). However, computing the CFM reduction of the TS, we do obtain the TS in (d). Mining for 1-bounded regions this latter system, we obtain a completely concurrent PN, that is a PN with transitions but without places.

## V. EXPERIMENTS

In the first set of experiments we will check the validity of the folding approach presented in Sect. IV by transforming some well-known logs from [15] into TSs. In particular we will use the following conversions

| Log | a12_1 | a12_5 | a22_1 | a22_5 | a32_1 | a32_5 | t32_1 | t32_5 | a42_1 | a42_5 |
|---|---|---|---|---|---|---|---|---|---|---|
| #cases | 200 | 1800 | 100 | 900 | 100 | 900 | 200 | 1800 | 100 | 900 |
| $\|\Sigma\|$ | 12 | 12 | 22 | 22 | 32 | 32 | 33 | 33 | 42 | 42 |
| $\|S_s\|$ | 25 | 25 | 1309 | 9867 | 2011 | 16921 | 7717 | 64829 | 2865 | 24366 |
| $\|S_{\text{PMs}}\|$ | 25 | 25 | 1225 | 8736 | 1961 | 16323 | 7508 | 62412 | 2825 | 23932 |
| $\|S_m\|$ | 18 | 18 | 751 | 3291 | 1378 | 5544 | 7167 | 50436 | 2568 | 15816 |
| $\|S_{\text{PMm}}\|$ | 18 | 18 | 673 | 2432 | 1338 | 4988 | 6963 | 48185 | 2527 | 15408 |
| $\|S_{t1}\|$ | 18 | 18 | 102 | 103 | 552 | 704 | 4043 | mem | 1918 | 7412 |
| $\|S_{p1}\|$ | 18 | 18 | 102 | 103 | 533 | 704 | 805 | 1000 | 1741 | 7279 |
| $\|S_c\|$ | 13 | 13 | 80 | 80 | 399 | 471 | 805 | 1000 | 1414 | 4934 |
| $T_s$ | 0 | 0 | 0 | 0.1 | 0 | 0.1 | 0.1 | 0.6 | 0 | 0.2 |
| $T_{\text{PMs}}$ | 0.3 | 0.3 | 0.2 | 0.9 | 0.2 | 0.7 | 0.6 | 4.2 | 0.3 | 0.9 |
| $T_m$ | 0 | 0 | 0 | 0.1 | 0 | 0.2 | 0.2 | 1.5 | 0.1 | 0.4 |
| $T_{\text{PMm}}$ | 0.3 | 0.3 | 0.2 | 0.9 | 0.2 | 0.8 | 0.7 | 5.1 | 0.4 | 1.1 |
| $T_{t1}$ | 0 | 0 | 0 | 0.1 | 0.1 | 0.4 | 0.3 | mem | 0.1 | 0.9 |
| $T_{p1}$ | 0 | 0 | 0 | 0.2 | 0.1 | 0.3 | 0.4 | 5.4 | 0.1 | 0.9 |
| $T_c$ | 0 | 0 | 0 | 0.1 | 0 | 0.2 | 0.2 | 2.0 | 0.1 | 0.6 |
| $\|B_s\|$, $\|B_m\|$, $\|B_{\text{PMs}}\|$, $\|B_{\text{PMm}}\|$ | 12 | 12 | 22 | 22 | 32 | 32 | 33 | 33 | 42 | 42 |
| $\|B_{t1}\|$ | 12 | 12 | 17 | 17 | 29 | 27 | 29 | mem | 38 | 37 |
| $\|B_{p1}\|$ | 12 | 12 | 17 | 17 | 27 | 27 | 27 | 27 | 37 | 37 |
| $\|B_c\|$ | 10 | 10 | 16 | 16 | 26 | 26 | 27 | 27 | 35 | 35 |

TABLE I
CONVERSION RESULTS FOR SOME LOGS FROM [15].

(in parenthesis we give the subscript used in the tables to identify each conversion): Sequential (subscript **s**), Multiset (subscript **m**), Folding conversion for 1-bounded discovery using the iterative version of the tandem repeat detection algorithm (subscript **t1**), idem but using the Parikh trie algorithm (subscript **p1**) and the CFM reduction (subscript **c**).

Since the popular tool ProM [32] allows modifications that bear some resemblance to the folding strategy, we also compare with the logs that result from applying the *ExactTandemRepeats* filter in ProM, and then performing either the sequential or the multiset conversions. The *ExactTandemRepeats* filter simply removes all tandem repeats and leaves a single instance of all the repetitions, thus it is much less general than the approach presented here. However, since in these experiments we focused on conversions for 1-bounded discovery, the comparison is still meaningful. Hence, besides the conversions enumerated before, we have two additional conversions in the tables: ProM filter and Sequential conversion (subscript **PMs**), and ProM filter and Multiset conversion (subscript **PMm**).

Table I shows some relevant information of the logs used in the experiments, as well as the results of the conversions. The names of benchmarks have been abbreviated for clarity, the original name is obtained by adding "f0n00" before the underscore, e.g. a12_1 identifies log a12f0n00_1. For each benchmark we give the number of sequences it contains (*#cases*), the number of different events present in the log ($|\Sigma|$) and, for each conversion method $x$: the number of states of the corresponding TS (rows $|S_x|$), the time required to build the TS by each type of conversion (rows $T_x$), and the size of the corresponding region basis (rows $|B_x|$).

All the results were obtained on a PC with an Intel Core Duo at 2.10Ghz and 2Gb of RAM, running the 2.6 Linux kernel.

In terms of state reduction, the conversion methods can be broadly classified in three categories. The first one comprises the sequential conversions (**s** and **PMs**) which yield the highest number of states. Note that ProM filters make very little difference. The first big reduction is achieved by the multiset-based conversion methods (**m** and **PMm**), and, again, ProM filters only provide a marginal reduction. Finally, the proposed methods and the CFM reduction achieve the smallest number of states, with improvements that are sometimes of an order of magnitude with respect to the multiset conversions.

Looking in more detail this last group of methods, as expected, the Parikh trie approach always outperforms the exact tandem repeat algorithm in terms of state reduction. In one particular example

| | becnhmarks: a12_1, a12_5 | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Time / Conv. | s, PMs | m, PMm, t1, p1, c | | | | | | | | | | | | |
| genet | 0.1 | 0.1 | | | | | | | | | | | | |
| rbminer | 0.1 | 0.1 | | | | | | | | | | | | |
| | becnhmark: a22_1 | | | | | | | becnhmark: a22_5 | | | | | | |
| Time / Conv. | s | PMs | m | PMm | t1, p1 | | c | s | PMs | m | PMm | t1, p1 | | c |
| genet | 340 | 360 | 94 | 37 | 0.2 | | 0.2 | time | time | 265 | 137 | 0.1 | | 0.1 |
| rbminer | 6 | 6 | 5 | 5 | 0.2 | | 0.1 | 33 | 29 | 13 | 10 | 0.2 | | 0.1 |
| | becnhmark: a32_1 | | | | | | | becnhmark: a32_5 | | | | | | |
| Time / Conv. | s | PMs | m | PMm | t1 | p1 | c | s | PMs | m | PMm | t1, p1 | | c |
| genet | mem | mem | mem | mem | 26 | 20 | 9 | time | time | time | time | 1.3 | | 0.7 |
| rbminer | 67 | 66 | 53 | 53 | 2 | 2 | 1.3 | 400 | 387 | 146 | 133 | 1.8 | | 1.4 |
| | becnhmark: t32_1 | | | | | | | becnhmark: t32_5 | | | | | | |
| Time / Conv. | s | PMs | m | PMm | t1 | p1, c | | s | PMs | m | PMm | t1 | p1, c | |
| genet | mem | mem | time | time | time | 61 | | time | time | time | time | – | 5 | |
| rbminer | 20 | 19 | 18 | 18 | 10 | 2 | | 146 | 141 | 114 | 109 | – | 2 | |
| | becnhmark: a42_1 | | | | | | | becnhmark: a42_5 | | | | | | |
| Time / Conv. | s | PMs | m | PMm | t1 | p1 | c | s | PMs | m | PMm | t1 | p1 | c |
| genet | mem | mem | mem | mem | mem | mem | mem | time | time | time | time | time | time | time |
| rbminer | 380 | 375 | 354 | 348 | 13 | 11 | 8 | 2354 | 2315 | 1574 | 1533 | 31 | 31 | 18 |

TABLE II
MINING OF LARGE LOGS. TIME LIMIT WAS SET TO 1 HOUR AND MEMORY LIMIT TO 1 GB.

| | ILP | | | genet | | rbminer | |
|---|---|---|---|---|---|---|---|
| Log | P/F | Time | App. | P/F | App. | P/F | App. |
| a12_1, a12_5 | 11/25 | 1 | 1.0 | 11/25 | 1.0 | 11/25 | 1.0 |
| a22_1 | 19/49 | 3 | 0.95 | 19/49 | 0.95 | 19/49 | 0.93 |
| a22_5 | 19/49 | 23 | 0.95 | 19/49 | 0.94 | 19/49 | 0.94 |
| a32_1 | 31/73 | 25 | 0.93 | 32/75 | 0.94 | 32/75 | 0.94 |
| a32_5 | 31/73 | 112 | 0.93 | 31/73 | 0.95 | 31/73 | 0.95 |
| t32_1 | 30/72 | 288 | 0.99 | 30/72 | 0.98 | 31/74 | 0.92 |
| t32_5 | 30/72 | 9208 | 0.99 | 30/72 | 0.98 | 30/72 | 0.92 |
| a42_1 | 44/109 | 154 | 1.0 | memout | | 52/131 | 1.0 |
| a42_5 | 44/101 | 1557 | 1.0 | timeout | | 46/107 | 1.0 |

TABLE III
QUALITY OF MINED NETS AND COMPARISON WITH A LANGUAGE-BASED MINER.

t32f0n00_5 the latter could not finish due to an explosion in the number of patterns of a given length. Since execution times for both algorithms are very similar, for these benchmarks, the Parikh trie is consistently a better option.

On the other hand, when comparing to the CFM reduction, the latter always yields smaller TSs. The reason for such phenomenon is the structure of the logs used, in which all sequences always have an initial and a final distinct event (thus they are complete sequences). This can cause, as a side-effect, the folding of unfolded cycles as shown in Lemma 2. However, the folding algorithms are still applicable when there is no final event, thus they can be used in more general settings.

In terms of discovery speed-ups related to the TS and basis reductions, Tables II and III show the running times needed by two region-based tools to mine the TSs obtained by the different conversions. Note that genet does not rely on the region basis, thus it only benefits from the state reduction. However, even with only half of the benefits, the improvements are dramatic. On the other hand rbminer takes full advantage of both benefits, which translates also in orders of magnitude speed-ups (for instance in the a32_5 and a42_5 benchmarks).

Table III contains a comparison with the running times of a language-based region tool (ILP [15]) implemented in the ProM suite. In many cases the tools based in the classical region theory have a similar performance when standard conversions are used, and only become much more competitive than

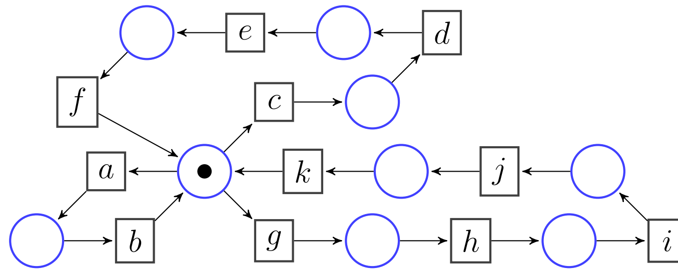



Fig. 14. A PN used in the experiments, namely `cy(3,2)`, containing a choice between three cycles of length 2, 3 and 4.

`ILP` when more advanced conversions, like the foldings or CFM reduction are used.

To compare the quality of the obtained nets using the most aggressive conversion (CFM), we give the number of places and arcs in the net (column *P/F*) and we also provide the well-known quality measure called *appropriateness* [33]. This metric quantifies to which extent the model describes the observed behavior, combined with the clarity degree of the model. It is normalized to be a real number between 0 (low) and 1 (high). Results show that the quality among all tools is basically the same, despite using an aggressive conversion like CFM.

However there are many scenarios in which the CFM reduction can yield inadmissible overapproximations. For instance if the knowledge that a sequence (a case in the log) has finished is not available. As an example of this kind of scenario, assume we want to derive the PN of a communication protocol from which we know only the beginning of some intercepted sequences. Since the sequences can stop at any arbitrary point of the protocol, we do not know whether the sequence is "complete". In these cases, CFM reduction might severely diminish the quality of the mined PN.

To illustrate this effect, we have run another set of experiments, in which cyclic PNs were used. For each net a log is created by randomly simulating the PN (Table IV). These logs are then converted into TSs and mined using the `rbminer` tool.

The first family of cyclic benchmarks, the `cy`$(m,n)$ benchmarks, corresponds to choices of $m$ cycles of increasing length, starting with a cycle of length $n$. Thus, `cy(3,2)` is a choice between cycles of length 2, 3 and 4. Fig. 14 shows this particular PN. In this case 5000 sequences were simulated, each one containing 50 events. The second family corresponds to a producer/consumer system, denoted `pc`$(m,n)$, with $m$ producers and $n$ consumers. Finally, the last family of cyclic benchmarks, namely `bp`$(n)$, represents a 2-bounded pipeline of $n$ processes.

Table IV details, for each benchmark, the number of sequences simulated (*#cases* row), the number of events in each sequence (*events/case*), the size of the alphabet of events ($|\Sigma|$) and the $k$-boundedness of the PN which originated the sequences. As before, we give the number of states of the TS obtained from the log using several conversion methods. For the folding methods, the $k$ used was the one corresponding to each benchmark. Next, the conversion times for each method are also provided, as well as the sizes of the resulting region bases. Finally, in the last two rows, we compare the running times of the `ILP` miner directly applied to the log, and `rbminer` when mining the resulting TS from the Parikh trie folding algorithm.

In all cases the PN mined from the CFM reduction contained no places. The PNs obtained by the `ILP` tool were only correct for the `cy` benchmarks, as the other types contain weighted arcs that `ILP` is not able to mine. For the `pc(8,5)` benchmark the tool could not complete the discovery process because it crashed due to a memory error in the LP solver.

On the other hand the PNs obtained by `rbminer` using the TSs from the Parikh trie folding algorithm were isomorphic to the ones used to generate the benchmarks. Moreover, the running times to mine such nets were several orders of magnitude smaller than the ones required by the `ILP` tool to obtain a net (in many cases lacking many places) from the log. These results show that the folding strategy using the Parikh trie is the only choice among the tested approaches that efficiently provides the correct nets.

| Log | cy(3,2) | cy(4,2) | cy(5,2) | pc(8,3) | pc(8,5) | bp(8) | bp(9) |
|---|---|---|---|---|---|---|---|
| #cases | 5000 | 5000 | 5000 | 5000 | 1000 | 100 | 100 |
| events/case | 50 | 50 | 50 | 50 | 250 | 5000 | 5000 |
| $\|\Sigma\|$ | 9 | 14 | 20 | 17 | 17 | 10 | 11 |
| $k$ | 1 | 1 | 1 | 3 | 5 | 2 | 2 |
| $\|S_s\|$ | 147398 | 157418 | 160240 | 233060 | 247385 | 498410 | 498317 |
| $\|S_m\|$ | 5289 | 18671 | 42264 | 196579 | 244484 | 13070 | 13094 |
| $\|S_{tf}\|$ | 7 | 11 | 16 | 196579 | 244484 | memout | memout |
| $\|S_{pf}\|$ | 7 | 11 | 16 | 1024 | 1536 | 1583 | 2976 |
| $\|S_c\|$ | 51 | 51 | 51 | 51 | 23 | 9457 | 11223 |
| $T_s$ | 1.2 | 1.2 | 1.2 | 1.4 | 1.4 | 3.0 | 2.8 |
| $T_m$ | 0.5 | 0.7 | 1.4 | 3.8 | 3.6 | 1.0 | 0.9 |
| $T_{tf}$ | 0.5 | 0.9 | 1.7 | 5.7 | 4.4 | memout | memout |
| $T_{pf}$ | 1.3 | 4.7 | 8.1 | 6.4 | 7.5 | 1.5 | 1.4 |
| $T_c$ | 0.6 | 0.8 | 1.7 | 5.1 | 4.5 | 1.0 | 1.1 |
| $\|B_m\|$ | 9 | 14 | 20 | 17 | 17 | 10 | 11 |
| $\|B_{tf}\|$ | 6 | 10 | 15 | 17 | 17 | memout | 11 |
| $\|B_{pf}\|$ | 6 | 10 | 15 | 9 | 9 | 8 | 9 |
| $\|B_c\|$ | 1 | 1 | 1 | 1 | 1 | 5 | 9 |
| `ILP` (s) | 7 | 89 | 853 | 7381 | error | 606 | 1518 |
| `rbminer` (s) | 0.1 | 0.1 | 0.3 | 0.1 | 0.1 | 0.1 | 0.3 |

TABLE IV
LOGS OBTAINED BY SIMULATION OF CYCLIC NETS.

# VI. CONCLUSIONS

A novel approach to convert a log into a transition system for process discovery has been presented, together with an algorithm to efficiently obtain such conversion. Experimental results show its validity to provide compact transition system that alleviate the complexity of deriving a PN from them.

## ACKNOWLEDGMENTS

This work has been supported by projects TIN2007-66523 and TIN2007-63927.

## REFERENCES

[1] W. van der Aalst, H. Reijers, and M. Song, "Discovering social networks from event logs," *Computer Supported Cooperative Work*, vol. 14, no. 6, pp. 549–593, 2005.
[2] W. van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 9, pp. 1128–1142, 2004.
[3] A. de Medeiros, W. van der Aalst, and A. Weijters, "Workflow mining: Current status and future directions," in *CoopIS/DOA/ODBASE*, 2003, pp. 389–406.
[4] L. Wen, W. van der Aalst, J. Wang, and J. Sun, "Mining process models with non-free-choice constructs," *Data Min. Knowl. Discov.*, vol. 15, no. 2, pp. 145–180, 2007.
[5] W. van der Aalst, A. de Medeiros, and A. Weijters, "Genetic process mining," in *ICATPN*, 2005, pp. 48–69.
[6] A. Ehrenfeucht and G. Rozenberg, "Partial (Set) 2-Structures. Part I, II," *Acta Informatica*, vol. 27, pp. 315–368, 1990.
[7] E. Badouel, L. Bernardinello, and P. Darondeau, "Polynomial algorithms for the synthesis of bounded nets," *Lecture Notes in Computer Science*, vol. 915, pp. 364–383, 1995.
[8] R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser, "Synthesis of petri nets from finite partial languages," *Fundam. Inform.*, vol. 88, no. 4, pp. 437–468, 2008.
[9] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev, "Deriving Petri nets from finite transition systems," *IEEE Transactions on Computers*, vol. 47, no. 8, pp. 859–882, Aug. 1998.
[10] J. Carmona, J. Cortadella, and M. Kishinevsky, "New region-based algorithms for deriving bounded Petri nets," *IEEE Transactions on Computers*, 2009.
[11] H.Verbeek, A.Pretorius, W. van der Aalst, and J. van Wijk, "On Petri-net synthesis and attribute-based visualization," in *Proc. Workshop on Petri Nets and Software Engineering (PNSE'07)*, Jun. 2007, pp. 127–141.
[12] 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 and Systems Modeling*, 2009.
[13] J. Carmona, J. Cortadella, and M. Kishinevsky, "A region-based algorithm for discovering Petri nets from event logs," in *6th International Conference on Business Process Management*, 2008, pp. 358–373.

[14] R. Bergenthum, J. Desel, R. Lorenz, and S.Mauser, “Process mining based on regions of languages,” in *Proc. 5th Int. Conf. on Business Process Management*, Sep. 2007, pp. 375–383.
[15] J. van der Werf, B. van Dongen, C. Hurkens, and A. Serebrenik, “Process discovery using integer linear programming,” in *Petri Nets*, 2008, pp. 368–387.
[16] J. Jansson and Z. Peng, “Online and dynamic recognition of squarefree strings,” *Mathematical Foundations of Computer Science 2005*, pp. 520–531, 2005.
[17] R. Kolpakov and G. Kucherov, “Finding maximal repetitions in a word in linear time,” in *In Proceedings of the 1999 Symposium on Foundations of Computer Science*. IEEE Computer Society, 1999, pp. 596–604.
[18] M. G. Main and R. J. Lorentz, “An o(n log n) algorithm for finding all repetitions in a string,” *Journal of Algorithms*, vol. 5, no. 3, pp. 422–432, 9 1984.
[19] A. Apostolico and F. P. Preparata, “Optimal off-line detection of repetitions in a string,” *Theoretical Computer Science*, vol. 22, no. 3, pp. 297–315, 2 1983.
[20] M. Crochemore, “Transducers and repetitions,” *Theoretical Computer Science*, vol. 45, pp. 63–86, 1986.
[21] D. Gusfield and J. Stoye, “Linear time algorithms for finding and representing all the tandem repeats in a string,” *Journal of Computer and System Sciences*, vol. 69, no. 4, pp. 525–546, 12 2004.
[22] M. Solé and J. Carmona, “Process mining from a basis of state regions,” in *Application and Theory of Petri Nets and other Models of Concurrency*, ser. LNCS. Springer, 2010, pp. 226–245.
[23] A. Arnold, *Finite Transition Systems*. Prentice Hall, 1994.
[24] T. Murata, “Petri Nets: Properties, analysis and applications,” *Proceedings of the IEEE*, pp. 541–580, Apr. 1989.
[25] J. Desel and W. Reisig, “The synthesis problem of Petri nets,” *Acta Inf.*, vol. 33, no. 4, pp. 297–315, 1996.
[26] M. Mukund, “Petri nets and step transition systems,” *Foundations of Comp. Science*, vol. 3, no. 4, pp. 443–478, 1992.
[27] L. Bernardinello, G. D. Michelis, K. Petruni, and S. Vigna, “On the synchronic structure of transition systems.” in *Structures in Concurrency Theory*, 1995, pp. 69–84.
[28] E. Badouel and P. Darondeau, “Theory of regions,” in *Petri Nets*, 1998, pp. 529–586.
[29] N. Busi and G. M. Pinna, “Process discovery and Petri nets,” *Mathematical Structures in Computer Science*, vol. 19, no. Special Issue 06, pp. 1091–1124, 2009.
[30] 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 and Systems Modeling*, 2009.
[31] M. Solé and J. Carmona, “rbminer:: A tool for discovering Petri nets from transition systems,” in *ATVA*, 2010.
[32] W. van der Aalst, B. van Dongen, C. Günther, R. Mans, A. de Medeiros, A. Rozinat, V. Rubin, M. Song, H. Verbeek, and A. Weijters, “ProM 4.0: Comprehensive support for *eal* process analysis,” in *ICATPN*, 2007, pp. 484–494.
[33] A. Rozinat and W. M. P. van der Aalst, “Conformance checking of processes based on monitoring real behavior,” *Inf. Syst.*, vol. 33, no. 1, pp. 64–95, 2008.