EINDHOVEN UNIVERSITY OF TECHNOLOGY

Process discovery using integer linear programming

Citation for published version (APA):
Werf, van der, J. M. E. M., Dongen, van, B. F., Hee, van, K. M., Hurkens, C. A. J., & Serebrenik, A. (2008). *Process discovery using integer linear programming*. (Computer science reports; Vol. 0804). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/2008

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 22. Aug. 2022

Process Discovery using Integer Linear Programming

J.M.E.M. van der Werf, B.F. van Dongen, K.M. van Hee,
C.A.J. Hurkens, and A. Serebrenik

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{ j.m.e.m.v.d.werf, b.f.v.dongen, k.m.v.hee,
c.a.j.hurkens, a.serebrenik }@tue.nl

Abstract. The research domain of *process discovery* aims at constructing a process model (e.g. a Petri net) which is an abstract representation of an execution log. Such a Petri net should (1) be able to reproduce the log under consideration and (2) be independent of the number of cases in the log. In this paper, we present a process discovery algorithm where we use concepts taken from the language-based theory of regions, a well-known Petri net research area. We identify a number of shortcomings of this theory from the process discovery perspective, and we provide solutions based on integer linear programming.

1 Introduction

Enterprise information systems typically log information on the steps performed by the users of the system. For legacy information systems, such execution logs are often the only means for gaining insight into ongoing processes. Especially, since system documentation is usually missing or obsolete and nobody is confident enough to provide such documentation. Hence, in this paper we consider the problem of *process discovery* [7], i.e. we construct a process model describing the processes controlled by the information system by simply using the execution log. We restrict our attention to the control flow, i.e., we focus on the ordering of activities executed, rather than on the data recorded.

Table 1 illustrates our notion of an event log, where it is important to realize that we assume that every event recorded is related to a single execution of a process, also referred to as a *case*.

A process model (in our case a Petri net) discovered from a given execution log should satisfy a number of requirements. First of all, such a Petri net should be capable of reproducing the log, i.e. every sequence of events recorded in the log should correspond to a firing sequence of the Petri net. Second, the size of the Petri net should be independent of the number of cases in the log. Finally, the Petri net should be such that the places in the net are as expressive as possible in terms of the dependencies between transitions they express.

A problem similar to process discovery arises in areas such as hardware design and control of manufacturing systems. There, the so called *theory of regions* is

Table 1. An event log.

case id	activity id	originator	case id	activity id	originator
case 1	activity A	John	case 5	activity A	Sue
case 2	activity A	John	case 4	activity C	Carol
case 3	activity A	Sue	case 1	activity D	Pete
case 3	activity B	Carol	case 3	activity C	Sue
case 1	activity B	Mike	case 3	activity D	Pete
case 1	activity C	John	case 4	activity B	Sue
case 2	activity C	Mike	case 5	activity E	Claire
case 4	activity A	Sue	case 5	activity D	Claire
case 2	activity B	John	case 4	activity D	Pete
case 2	activity D	Pete			

used to construct a Petri net from a behavioral specification (e.g., a language), such that the behavior of this net corresponds with the specified behavior (if such a net exists).

In this paper we investigate the application of the theory of regions in the field of process discovery. It should be noted that we are not interested in Petri nets whose behavior corresponds completely with the given execution log, i.e. logs cannot be assumed to exhibit all behavior possible. Instead, they merely provide insights into “common practice” within a company.

In Section 3, we show that a straightforward application of the theory of regions to process discovery would lead to Petri nets of which the number of places depends on size of the log. Therefore, in Section 4, we discuss how ideas from the theory of regions can be combined with generally accepted concepts from the field of process discovery to generate Petri nets that satisfy the requirements given above. Furthermore, using the log of Table 1 as an illustrative example, we show how our approach can lead to Petri nets having certain structural properties, such as marked graphs, state machines and free-choice nets [14]. Finally, in Section 5 we present the implementation of our approach in ProM [4], followed by a brief discussion on its usability on logs taken from practice. In Section 6, we provide some conclusions.

2 Preliminaries

Let S be a set. The powerset of S, the set of all subsets of S, is denoted by $\mathcal{P}(S) = \{S'|S' \subseteq S\}$. A *bag (multiset)* m over S is a function $S \rightarrow \mathbb{N}$, where $\mathbb{N} = \{0, 1, 2, \ldots\}$ denotes the set of natural numbers. The set of all bags over S is denoted by $\mathbb{N}^S$. We identify a bag with all elements occurring only once with the set containing these elements, and vice versa. We use $+$ and $-$ for the sum and difference of two bags, and $=, <, >, \leq, \geq$ for the comparison of two bags, which are defined in a standard way. We use $\emptyset$ for the empty bag, and $\in$ for the element inclusion. We write e.g. $m = 2[p] + [q]$ for a bag m with $m(p) = 2$, $m(q) = 1$ and $m(x) = 0$, for all $x \notin \{p, q\}$. We use the standard notation $|m|$ and $|S|$ to denote

the number of elements in bags and sets. Let $n \in \mathbb{N}$. A *sequence* over S of length n is a function $\sigma \colon \{1, \ldots, n\} \to S$. If $n > 0$ and $\sigma(1) = a_1, \ldots, \sigma(n) = a_n$, we write $\sigma = \langle a_1, \ldots, a_n \rangle$, and σ_i for $\sigma(i)$. The length of a sequence is denoted by $|\sigma|$. The sequence of length 0 is called the empty sequence, and is denoted by ϵ. The set of finite sequences over S is denoted by S^*. Let $\upsilon, \tau \in S^*$ be two sequences. Concatenation, denoted by $\sigma = \upsilon; \tau$ is defined as $\sigma : \{1, \ldots, |\upsilon| + |\tau|\} \to S$, such that for $1 \leq i \leq |\upsilon|$, $\sigma(i) = \upsilon(i)$, and for $|\upsilon| + 1 \leq i \leq |\sigma|$, $\sigma(i) = \tau(i - |\upsilon|)$. Further, we define the prefix $\leq$ on sequences by $\upsilon \leq \tau$ if and only if there exists a sequence $\rho \in S^*$ such that $\tau = \upsilon; \rho$. We use $\boldsymbol{x}$ to denote column vectors and for a sequence $\sigma \in S^*$, the *Parikh vector* $\boldsymbol{\sigma} \colon S \to \mathbb{N}$ defines the number of occurrences of each element of S in the sequence, i.e. $\boldsymbol{\sigma}(s) = |\{i | 1 \leq i \leq |\sigma|, \sigma(i) = s\}|$, for all $s \in S$.

Let Σ be an *alphabet*, i.e. a finite and non-empty set. Its elements are called *letters*. A *word* w is a finite sequence of letters, i.e. $w \in \Sigma^*$. A *language* $\mathcal{L}$ is a set of words: $\mathcal{L} \subseteq \Sigma^*$. A language is called *prefix-closed* if for all non-empty words $w = w'a \in \mathcal{L}$, $a \in \Sigma$, it holds that $w' \in \mathcal{L}$.

Definition 2.1. (Petri net) A *Petri net* N is a 3-tuple $N = (P, T, F)$, where (1) P and T are two disjoint sets of *places* and *transitions* respectively; we call the elements of the set $P \cup T$ nodes of N; (2) $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*; an element of F is called an *arc*.

Let $N = (P, T, F)$ be a Petri net. Given a node $n \in P \cup T$, we define its preset $\overset{\bullet}{_N}n = \{n' | (n', n) \in F\}$, and its postset $n^{\bullet}_N = \{n' | (n, n') \in F\}$. If the context is clear, we omit the N in the subscript.

To describe the execution semantics of a net, we use *markings*. A marking m of a net $N = (P, T, F)$ is a bag over P. Markings are states (configurations) of a net. A pair (N, m) is called a *marked Petri net*. A transition $t \in T$ is *enabled* in a marking $m \in \mathbb{N}^P$, denoted by $(N, m)[t\rangle$ if and only if ${}^{\bullet}t \leq m$. Enabled transitions may *fire*. A transition firing results in a new marking m' with $m' = m - {}^{\bullet}t + t^{\bullet}$, denoted by $(N, m)\ [t\rangle\ (N, m')$.

Definition 2.2. (Firing sequence) Let $N = (P, T, F)$ be a Petri net, and (N, m) be a marked Petri net. A sequence $\sigma \in T^*$ is called a *firing sequence* of (N, m) if and only if for $n = |\sigma|$, there exist markings $m_1, \ldots, m_{n-1} \in \mathbb{N}^P$ and transitions $t_1, \ldots, t_n \in T$ such that $\sigma = \langle t_1, \ldots, t_n \rangle$, and, $(N, m)\ [t_1\rangle\ (N, m_1) \ldots (N, m_{n-1})[t_n\rangle$. We lift the notations for being enabled and firing to firing sequences, i.e. if $\sigma \in T^*$ be a firing sequence, then for all $1 \leq k \leq |\sigma|$ and for all places $p \in P$ it holds that $m(p) + \sum_{i=1}^{k-1} (\sigma_i{}^{\bullet}(p) - {}^{\bullet}\sigma_i(p)) \geq {}^{\bullet}\sigma_k(p)$.

As we stated before, a single execution of a model is called a case. If the model is a Petri net, then a single firing sequence that results in a dead marking (a marking where no transition is enabled) is a case. Since most information systems log all kinds of events during execution of a process, we establish a link between an execution log of an information system and firing sequences of Petri nets. The basic assumption is that the log contains information about specific *transitions* executed for specific *cases*.

Definition 2.3. (Case, Log) Let T be a set of transitions, $\sigma \in T^*$ is a *case*, and $L \in \mathcal{P}(T^*)$ is an *execution log* if and only if for all $t \in T$ holds that there is a $\sigma \in L$, such that $t \in \sigma$. In other words, an execution log is a set of firing sequences of some marked Petri net $(N, m) = ((P, T, F), m)$, where P, F and m are unknown and where no transition is dead.

In Definition 2.3, we define a log as a *set* of cases. Note that in real life, logs are *bags* of cases, i.e. a case with a specific order in which transitions are executed may occur more than once, as shown in our example. However, in this paper, we do not have to consider occurrence frequencies of cases and therefore sets suffice.

Recall that the goal of process discovery is to obtain a Petri net that can reproduce the execution log under consideration. In [11], Lemma 1 states the conditions under which this is the case. Here, we repeat that Lemma and we adapt it to our own notation.

Definition 2.4. (Replayable log) Let $L \in \mathcal{P}(T^*)$ be an execution log, and $\sigma \in L$ a case. Furthermore, let $N = ((P, T, F), m)$ be a marked Petri net. If the log L is a subset of all possible cases of (N, m), i.e. each case in L is a firing sequence in (N, m) ($(N, m)[\sigma\rangle$), we say that L can be replayed by (N, m).

In order to construct a Petri net that can indeed reproduce a given log, the theory of regions can be used. In Section 3, we present this theory in detail and we argue why the classical algorithms in existence are not directly applicable in the context of process discovery. In Section 4, we show how to extend the theory of regions to be more applicable in a process discovery context.

3 Theory of regions

The general question answered by the theory of regions is: given the specified behavior of a system, what is the Petri net that represents this behavior? Both the form in which the behavior is specified as well as the "represents" statement can be expressed in different ways. Mainly, we distinguish two types, the first of which is state-based region theory [9,12,16]. This theory focusses on the synthesis of Petri nets from state-based models, where the statespace of the Petri net is branching bisimilar to the given state-based model. Although state-based region theory can be applied in the process discovery context [6], the main problem is that execution logs rarely carry state information and the construction of this state information from a log is far from trivial [6].

In this paper, we consider language-based region theory [8, 13, 17], of which [17] presents a nice overview. In [17], the authors show how for different classes of languages (step languages, regular languages and partial languages) a Petri net can be derived such that the resulting net is the smallest Petri net in which the words in the language are possible firing sequences.

In this section, we introduce the language-based regions, we briefly show how the authors of [11, 17] used these regions in the context of process discovery and why we feel that this application is not a suitable one.

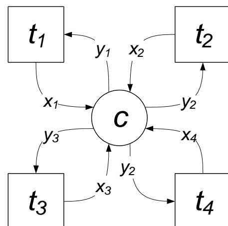

Fig. 1. Region for a log with four events.

3.1 Language-based theory of regions

Given a prefix-closed language $\mathcal{L}$ over some alphabet T, the language-based theory of regions tries to find a finite Petri net $N(\mathcal{L})$ in which the transitions correspond to the symbols in the alphabet of the language and of which all words in the language are firing sequences. Furthermore, the Petri net should have only minimal firing sequences not in the language.

The Petri net $N(\mathcal{L}) = (\emptyset, T, \emptyset)$ is a finite Petri net in which all words are firing sequences. However, its behavior not minimal. Therefore, the behavior of this Petri net needs to be reduced, such that the Petri net still allows to reproduce all words in the language, but does not allow for more behavior. This is achieved by adding places to the Petri net. The theory of regions provides a method to calculate these places, using *regions*.

Definition 3.1. (Region) A region of a prefix-closed language $\mathcal{L}$ over $T = \{t_1, \ldots, t_n\}$ is a triple $(\boldsymbol{x}, \boldsymbol{y}, c)$ with $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^{|T|}$ and $c \in \{0,1\}$, such that for each non-empty word $w = w'; a \in \mathcal{L}$, $w' \in \mathcal{L}$, $a \in T$:

$$c + \sum_{1 \leq i \leq n} (\boldsymbol{w'}(t_i) \cdot \boldsymbol{x}(t_i) - \boldsymbol{w}(t_i) \cdot \boldsymbol{y}(t_i)) \geq 0$$

This can be rewritten into the inequation system:

$$\mathbf{1} \cdot c + A' \cdot \boldsymbol{x} - A \cdot \boldsymbol{y} \geq \mathbf{0}$$

where A and A' are two $|\mathcal{L}| \times |T|$ matrices with $A(w,t) = \boldsymbol{w}(t)$, and $A'(w,t) = \boldsymbol{w'}(t)$, with $w = w'a$. The set of all regions of a language is denoted by $\Re(\mathcal{L})$ and the region $(\mathbf{0}, \mathbf{0}, 0)$ is called the *trivial region*.[1]

Figure 1 shows a region for a log with four events, i.e. each solution $(\boldsymbol{x}, \boldsymbol{y}, c)$ of the inequation system can be regarded in the context of a Petri net, where the region corresponds to a feasible place with preset $\{t | t \in T, \boldsymbol{x}(t) = 1\}$ and postset $\{t | t \in T, \boldsymbol{y}(t) = 1\}$, and initially marked with c tokens. Note that we do not assume arc-weights, where the authors of [8, 11, 13, 17] do. However, in process

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

modelling languages, such arc weights typically do not exist, hence we decided to ignore them. Our approach can however easily be extended to incorporate them.

Since the place represented by a region is a place which can be added to a Petri net, without disturbing the fact that the net can reproduce the language under consideration, such a place is called a *feasible* place.

Definition 3.2. (Feasible place) Let $\mathcal{L}$ be a prefix-closed language over T and let $N = ((P, T, F), m)$ be a marked Petri net. A place $p \in P$ is is called *feasible* if and only if there exists a *corresponding* region $(\boldsymbol{x}, \boldsymbol{y}, c) \in \Re(\mathcal{L})$ such that $m(p) = c$, and $\boldsymbol{x}(t) = 1$ if and only if $t \in {}^{\bullet}p$, and $\boldsymbol{y}(t) = 1$ if and only if $t \in p^{\bullet}$.

In [11, 17] it was shown that any solution of the inequation system of Definition 3.1 can be added to a Petri net without influencing the ability of that Petri net to replay the log. However, since there are infinitely many solutions of that inequation system, there are infinite many feasible places and the authors of [11, 17] present two ways of finitely representing these places.

Basis representation In the basis representation, the set of places is chosen such that it is a basis for the non-negative integer solution space of the linear inequation system. Although such a basis always exists for homogeneous inequation systems, it is worst-case exponential in the number of equations [17]. By construction of the inequation system, the number of equations is linear in the number of traces, and thus, the basis representation is worst-case exponential in the number of events in the log. Hence, an event log containing ten thousand events, referring to 40 transitions, might result in a Petri net containing a hundred million places connected to those 40 transitions. Although [11] provides some ideas on how to remove redundant places from the basis, these procedures still require the basis to be constructed fully. Furthermore, the implementation of the work in [11] is not publicly available for testing and no analysis is presented of this approach on realistically sized logs.

Separating representation To reduce the theoretical size of the resulting Petri net, the authors of [11,17] propose a separating representation. In this representation, places that separate the allowed behavior (specified by the system) using words not in the language are added to the resulting Petri net. Although this representation is no longer exponential in the size of the language, but polynomial, it requires the user to specify undesired behavior, which can hardly be expected in the setting of process discovery, i.e. nothing is known about the behavior, except the behavior seen in the event log. Furthermore, again no analysis is presented of this approach on realistically sized logs.

In [11, 17] the authors propose the separating representation for regular and step languages and by this, they maximize the number of places generated by the number of events in the log times the number of transitions. As we stated in

the introduction, the size of the Petri net should not depend on the size of the log for the approach to be applicable in the context of process discovery.

4 Integer Linear Programming formulation

In [11, 17] it was shown that any solution of the inequation system of Definition 3.1 can be added to a Petri net without influencing the ability of that Petri net to replay the log. Both the basis and the separating representation are presented to select which places to indeed add to the Petri net. However, as shown in Section 3, we argue that the theoretical upper bound on the number of places selected is high. Therefore, we take a different selection mechanism for adding places that:

- Explicitly express certain causal dependencies between transitions that can be discovered from the log, and
- Favors places which are more expressive than others (i.e. the added places restrict the behavior as much as possible).

In this section, we first show how to express a log as a prefix-closed language, which is a trivial, but necessary step and we quantify the expressiveness of places, in order to provide a target function, necessary to translate the inequation system of Definition 3.1 into a integer linear programming problem in Subsection 4.2. In Section 4.3, we show a first algorithm to generate a Petri net. In Subsection 4.4, we then provide insights into the causal dependencies found in a log and how these can be used for finding places. We conclude this section with a description of different algorithms for different classes of Petri nets.

4.1 Log to Language

To apply the language-based theory of regions in the field of process discovery, we need to represent the process log as a prefix-closed language, i.e. by all the traces present in the process log, and their prefixes. Recall from Definition 2.3 that a process log is a finite set of traces.

Definition 4.1. (Language of a process log) Let T be a set of activities, $L \in \mathcal{P}(T^*)$ a process log over this set of transitions. The language $\mathcal{L}$ that represents this process log, uses alphabet T, and is defined by:

$$\mathcal{L} = \{l \in T^* | \exists l' \in L \colon l \leq l'\}$$

As mentioned before, a trivial Petri net capable of reproducing a language is a net with only transitions. To restrict the behavior allowed by the Petri net, but not observed in the log, we start adding places to that Petri net. However, the places that we add to the Petri net should be *as expressive as possible*, which can be expressed using the following observation. If we remove the arc (p, t) from F in a Petri net $N = (P, T, F)$ (assuming $p \in P, t \in T, (p, t) \in F$), the resulting net still can replay the log (as we only weakened the pre-condition of transition

t). Also if we would add a non-existing arc (t, p) to F, with $t \in T$ and $p \in P$, the resulting net still can replay the log as it strengthens the post-condition of t.

Lemma 4.2. (Adding an incoming arc to a place retains behavior) Let $N = ((P, T, F), m)$ be a marked Petri net that can replay the process log $L \in \mathcal{P}(T^*)$. Let $p \in P$ and $t \in T$ such that $(t, p) \notin F$. The marked Petri nets $N' = ((P, T, F'), m)$ with $F' = F \cup \{(t, p)\}$ can replay the log L.

Proof. Let $\sigma = \sigma_1; t; \sigma_2 \in T^*$, such that $t \notin \sigma_1$ be a firing sequence of (N, m). Let $m' \in \mathbb{N}^P$ such that $(N, m)\,[\sigma_1; t\rangle\,(N, m')$ and $(N, m')[\sigma_2\rangle$ We know that for all $p' \in t^\bullet_N$ holds that $m'(p') > 0$, since t just fired. Assume $t \notin \sigma_1$. Then $(N', m)\,[\sigma_1; t\rangle\,(N', m'')$ with $m'' = m' + [p]$. Furthermore, since $m' + [p] > m'$, we know that $(N', m'')[\sigma_2\rangle$. By induction on the occurrences of t, we get $(N', m)[\sigma\rangle$. □

Lemma 4.3. (Removing an outgoing arc from a place retains behavior) Let $N = ((P, T, F), m)$ be a marked Petri net that can replay the process log $L \in \mathcal{P}(T^*)$. Let $p \in P$ and $t \in T$ such that $(p, t) \in F$. The marked Petri nets $N' = ((P, T, F'), m)$ with $F' = F \setminus \{(p, t)\}$ can replay the log L.

Proof. Let $\sigma = \sigma_1; t; \sigma_2 \in T^*$, such that $t \notin \sigma_1$ be a firing sequence of (N, m). Let $m' \in \mathbb{N}^P$ such that $(N, m)\,[\sigma_1\rangle\,(N, m')$. We know that for all $p' \in^\bullet_N t$ holds that $m'(p') > 0$, since t is enabled. This implies that for all $p' \in^\bullet_{N'} t$ also holds that $m'(p') > 0$, since ${}^\bullet_{N'} t = {}^\bullet_N t \setminus \{p\}$. Hence, $(N', m)\,[\sigma_1\rangle\,(N', m')$ and $(N', m')\,[t\rangle\,(N', m'')$. Due to monotonicity, we have $(N', m'')[\sigma_2\rangle$. Hence $(N', m)[\sigma\rangle$.

□

Besides searching for regions that lead to places with maximum expressiveness, i.e. a place with a maximum number of input arcs and a minimal number of output arcs, we are also searching for "minimal regions". As in the region theory for synthesizing Petri nets with arc weights, minimal regions are regions that are not the sum of two other regions.

Definition 4.4. (Minimal region) Let $L \in \mathcal{P}(T^*)$ be a log, let $r = (\boldsymbol{x}, \boldsymbol{y}, c)$ be a region. We say that r is *minimal*, if there do not exist two other, non trivial, regions r_1, r_2 with $r_1 = (\boldsymbol{x_1}, \boldsymbol{y_1}, c_1)$ and $r_2 = (\boldsymbol{x_2}, \boldsymbol{y_2}, c_2)$, such that $\boldsymbol{x} = \boldsymbol{x_1} + \boldsymbol{x_2}$ and $\boldsymbol{y} = \boldsymbol{y_1} + \boldsymbol{y_2}$ and $c = c_1 + c_2$.

Using the inequation system of Definition 3.1, the expressiveness of a place and the partial order on regions, we can define an integer linear programming problem (ILP formulation [20]) to construct the places of a Petri net in a logical order.

4.2 ILP formulation

In order to transform the linear inequation system introduced in Section 3 to an ILP problem, we need a target function. Since we have shown that places are

most expressive if their input is minimal and their output is maximal, we could minimize the value of a target function $f((\boldsymbol{x}, \boldsymbol{y}, c)) = c + \mathbf{1}^\mathsf{T} \cdot \boldsymbol{x} - \mathbf{1}^\mathsf{T} \cdot \boldsymbol{y}$, i.e. adding an input arc from the corresponding feasible place increases the value of $\mathbf{1}^\mathsf{T} \cdot \boldsymbol{x}$, as does removing an output arc from the corresponding feasible place.

However, it is easy to see that this function does not favor minimal regions, i.e. regions that are no sum of two other regions. Therefore, we need a target function f', such that for any three regions r_1, r_2 and $r3$ with $r_3 = r_1 + r + 2$ holds that $f'(r_1) < f'(r_3)$ and $f'(r_2) < f'(r_3)$, while preserving the property that regions corresponding to more expressive feasible places have a lower target value. For this purpose, we use the target function $f'((\boldsymbol{x}, \boldsymbol{y}, c)) = c + \mathbf{1}^\mathsf{T} \cdot A \cdot \boldsymbol{x} - \mathbf{1}^\mathsf{T} \cdot A \cdot \boldsymbol{y}$.

Definition 4.5. (Target function) Let $L \in \mathcal{P}(T^*)$ be a log, let $r = (\boldsymbol{x}, \boldsymbol{y}, c)$ be a region and let A be the matrix as in definition 3.1. We define a target function $\tau : \Re(\mathcal{L}) \to \mathbb{N}$, such that $\tau(r) = c + \mathbf{1}^\mathsf{T} \cdot A \cdot (\boldsymbol{x} - \boldsymbol{y})$.

First, we show that the target function of Definition 4.5 indeed satisfies all criteria.

Lemma 4.6. (Target function favors minimal regions) Let $L \in \mathcal{P}(T^*)$ be a log, let $r_1 = (\boldsymbol{x_1}, \boldsymbol{y_1}, c_1)$, $r_2 = (\boldsymbol{x_2}, \boldsymbol{y_2}, c_2)$ and $r_3 = (\boldsymbol{x_1} + \boldsymbol{x_2}, \boldsymbol{y_1} + \boldsymbol{y_2}, c_1 + c_2)$ be three regions (i.e. with $r_3 = r_1 + r_2$). Furthermore, let A be the matrix defined in Definition 3.1. Let $\tau : \Re(\mathcal{L}) \to \mathbb{N}$, such that $\tau(r) = c + \mathbf{1}^\mathsf{T} \cdot A \cdot (\boldsymbol{x} - \boldsymbol{y})$. Then $\tau(r_1) < \tau(r_3)$ and $\tau(r_2) < \tau(r_3)$.

Proof. First, we show that $\tau(r_3) = \tau(r_1) + \tau(r_2)$. Clearly $\tau(r_3) = c_1 + c_2 + \mathbf{1}^\mathsf{T} \cdot A \cdot (\boldsymbol{x_1} + \boldsymbol{x_2} - (\boldsymbol{y_1} + \boldsymbol{y_2}))$ and hence $\tau(r_3) = c_1 + \mathbf{1}^\mathsf{T} \cdot A \cdot (\boldsymbol{x_1} - \boldsymbol{y_1}) + c_2 + \mathbf{1}^\mathsf{T} \cdot A \cdot (\boldsymbol{x_2} - \boldsymbol{y_2}) = \tau(r_2) + \tau(r_3)$.

It remains to be shown that for any region $r = (\boldsymbol{x}, \boldsymbol{y}, c)$ holds that $\tau(r) > 0$. Recall from Definition 3.1 that $c + \mathbf{1}^\mathsf{T} \cdot A' \cdot \boldsymbol{x} - \mathbf{1}^\mathsf{T} \cdot A \cdot \boldsymbol{y} \geq 0$. Since we know that for all $t \in T$, there is a $\sigma \in L$ with $t \in \sigma$ (Definition 2.3), we know that $\mathbf{1}^\mathsf{T} \cdot A > \mathbf{0}$ and $\mathbf{1}^\mathsf{T} \cdot A' \geq \mathbf{0}$ (the sum over the columns of A produces the total number of occurrences of each transition in the language and in A' the last occurrence in each prefix is not counted). Hence, $\mathbf{1}^\mathsf{T} \cdot A \geq \mathbf{1}^\mathsf{T} \cdot A'$ and therefore $c + \mathbf{1}^\mathsf{T} \cdot A \cdot \boldsymbol{x} - \mathbf{1}^\mathsf{T} \cdot A \boldsymbol{y} \geq c + \mathbf{1}^\mathsf{T} \cdot A' \cdot \boldsymbol{x} - \mathbf{1}^\mathsf{T} \cdot A \boldsymbol{y} \geq 0$. Assuming that $\mathbf{1}^\mathsf{T} \cdot \boldsymbol{x} + \mathbf{1}^\mathsf{T} \cdot \boldsymbol{y} > 0$ (i.e. the place corresponding to region r has at least one connecting arc), we can derive that $c + \mathbf{1}^\mathsf{T} \cdot A \cdot \boldsymbol{x} - \mathbf{1}^\mathsf{T} \cdot A \boldsymbol{y} \neq 0$ and hence we have shown that $\tau(r) > 0$. Combining this with the fact that $\tau(r_3) = \tau(r_1) + \tau(r_2)$, we have shown that $\tau(r_3) > \tau(r_1)$ and $\tau(r_3) > \tau(r_2)$. □

Lemma 4.6 shows that our target function satisfies our requirements provided that each region translates into a place with at least one arc, incoming or outgoing. This does not restrict the generality of our approach, since places without arcs attached to them are of no interest for the behavior of a Petri net.

Combining Definition 3.1 with the condition set by Lemma 4.6 and the target function of Definition 4.5, we get the following integer linear programming problem.

Definition 4.7. (ILP formulation) Let $L \in \mathcal{P}(T^*)$ be a log, and let A and A' be the matrices defined in Definition 3.1. We define the ILP ILP_L corresponding

with this log as:

$$
\begin{array}{lll}
\text{Minimize } c + \mathbf{1}^\mathsf{T} \cdot A \cdot (\boldsymbol{x} - \boldsymbol{y}) & & \text{Defintion 4.5} \\
\text{such that} \quad & c + A' \cdot \boldsymbol{x} - A \cdot \boldsymbol{y} \geq \mathbf{0} & \text{Definition 3.1} \\
& \mathbf{1}^\mathsf{T} \cdot \boldsymbol{x} + \mathbf{1}^\mathsf{T} \cdot \boldsymbol{y} \geq 1 & \text{There should be at least one edge} \\
& \mathbf{0} \leq \boldsymbol{x} \leq \mathbf{1} & x \in \{0,1\}^{|T|} \\
& \mathbf{0} \leq \boldsymbol{y} \leq \mathbf{1} & y \in \{0,1\}^{|T|} \\
& 0 \leq c \leq 1 & c \in \{0,1\}
\end{array}
$$

The ILP problem presented in Definition 4.7 provides the basis for our process discovery problem. However, an optimal solution to this ILP only provides a single feasible place with a minimal value for the target function. Therefore, in the next subsection, we show how this ILP problem can be used as a basis for constructing a Petri net from a log.

4.3 Constructing Petri nets using ILP

In the previous subsection, we provided the basis for constructing a Petri net from a log. In fact, the target function of Definition 4.5 provides a partial order on all elements of the set $\Re(\mathcal{L})$, i.e. the set of all regions of a language. In this subsection, we show how to generate the first n places of a Petri net, that is (1) able to reproduce a log under consideration and (2) of which the places are as expressive as possible.

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

Definition 4.8. (Refining the ILP after each solution) Let $L \in \mathcal{P}(T^*)$ be a log, let A and A' be the matrices defined in Definition 3.1 and let $ILP_{(L,0)}$ be the corresponding ILP. Furthermore, let region $r_0 = (\boldsymbol{x_0}, \boldsymbol{y_0}, c_0)$ be a minimal solution of $ILP_{(L,0)}$. We define the refined ILP as $ILP_{(L,1)}$, with the extra constraint specifying that:

$$-c_0 \cdot c + \boldsymbol{y}^\mathsf{T} \cdot (\mathbf{1} - \boldsymbol{y_0}) - \boldsymbol{x}^\mathsf{T} \cdot \boldsymbol{x_0} \geq -c_0 + 1 - \mathbf{1}^\mathsf{T} \cdot \boldsymbol{x_0}$$

Lemma 4.9. (Refining yields unrelated regions) Let $L \in \mathcal{P}(T^*)$ be a log, let A and A' be the matrices defined in Definition 3.1 and let $ILP_{(L,0)}$ be the corresponding ILP. Furthermore, let region $r_0 = (\boldsymbol{x_0}, \boldsymbol{y_0}, c_0)$ be a minimal solution of $ILP_{(L,0)}$ and let $r_1 = (\boldsymbol{x_1}, \boldsymbol{y_1}, c_1)$ be a minimal solution of $ILP_{(L,1)}$, where $ILP_{(L,1)}$ is the refinement of $ILP_{(L,0)}$ following Definition 4.8. Then, $c_1 < c_0$ or there exists a $t \in T$, such that $x_1(t) < x_0(t) \vee y_1(t) > y_0(t)$.

Proof. Assume that $c_1 \geq c_0$ and for all $t \in T$ holds that $x_1(t) \geq x_0(t)$, and $y_1(t) \leq y_0(t)$. Then since $c_0, c_1 \in \{0,1\}$ we know that $c_0 \cdot c_1 = c_0$. Similarily,

$\boldsymbol{y_1}^\intercal \cdot (\mathbf{1} - \boldsymbol{y_0}) = 0$ and $\boldsymbol{x_0}^\intercal \cdot \boldsymbol{x_1} = \boldsymbol{x_0}^\intercal \cdot \mathbf{1}$. Hence $-c_0 \cdot c_1 + \boldsymbol{y_1}^\intercal \cdot (\mathbf{1} - \boldsymbol{y_0}) - \boldsymbol{x_0}^\intercal \cdot \boldsymbol{x_1} = -c_0 - \boldsymbol{x_0}^\intercal \cdot \mathbf{1}$ and since $-c_0 - \boldsymbol{x_0}^\intercal \cdot \mathbf{1} < -c_0 + 1 - \mathbf{1}^\intercal \cdot \boldsymbol{x_0}$, we know that r_1 is not a solution of $ILP_{(l,0)}$. This is a contradiction. □

The refinement operator presented above, basically defines an algorithm for constructing the places of a Petri net that is capable of reproducing a given log. The places are generated in an order which ensures that the most expressive places are found first and that only places are added that have less tokens, less outgoing arcs, or more incoming arcs. Furthermore, it is easy to see that the solutions of each refined ILP are also solutions of the original ILP, hence all places constructed using this procedure are feasible places.

The procedure, however, still has the downside that the total number of places introduced is worst-case exponential in the number of transitions. Furthermore, the first n places might be introduced linking a small number of transitions, whereas other transitions in the net are only linked after the first n places are found. Since there is no way to provide insights into the value of n for a given Petri net, we propose a more suitable approach, *not using the refinement step of Definition 4.8*. Instead, we propose to guide the search for solutions (i.e. for places) by metrics from the field of process discovery [5, 7, 15, 21].

4.4 Using log-based properties

Recall from the beginning of this section, that we are specifically interested in places expressing explicit causal dependencies between transitions. In this subsection, we first introduce how these causal dependencies are usually derived from a log file. Then we use these relations in combination with the ILP of Definition 4.7 to construct a Petri net.

Definition 4.10. (Causal dependency [7]) Let T be a set of transitions and $L \in \mathcal{P}(T^*)$ an execution log. If for two activities $a, b \in T$, there are traces $\sigma_1, \sigma_2 \in T^*$ such that $\sigma_1; a; b; \sigma_2 \in L$, we write $a >_L b$. If in a log L we have $a >_L b$ and not $b >_L a$, there is a *causal dependency* between a and b, denoted by $a \rightarrow_L b$.

In [7], it was shown that *if* a log L satisfies a certain completeness criterion and *if* there exists a Petri net of a certain class [7] that can reproduce this log, then the $>_L$ relation is enough to reconstruct this Petri net from the log. However, the completeness criterion assumes knowledge of the Petri net used to generate the log and hence it is undecidable whether an arbitrary log is complete or not. Nonetheless, we provide the formal definition of the notion of completeness, and we prove that for complete logs, causal dependencies directly relate to places and hence provide a good guide for finding these places.

Definition 4.11. (Complete log [7]) Let $N = ((P, T, F), m)$ be a marked Petri net. Let $L \in \mathcal{P}(T^*)$ be a process log. The log L is called *complete* if and only if there are traces $\sigma_1, \sigma_2 \in T^*$ such that $(N, m)[\sigma_1; a; b; \sigma_2\rangle$ implies $a >_L b$.

In [7], the proof that a causal dependency corresponds to a place is only given for safe Petri nets (where each place will never contain more than one token during execution). This can however be generalized for non-safe Petri nets.

Lemma 4.12. (Causality implies a place) Let $N = ((P, T, F), m)$ be a marked Petri net. Let L be a complete log of N. For all $a, b \in T$, it holds that if $a \neq b$ and $a \rightarrow_L b$ then $a^\bullet \cap {}^\bullet b \neq \emptyset$.

Proof. Assume $a \rightarrow_L b$ and $a^\bullet \cap {}^\bullet b = \emptyset$. By the definition of $>_L$, there exist sequences $\sigma_1, \sigma_2 \in T^*$ such that $(N, m)[\sigma_1; a; b; \sigma_2\rangle$. Let $s = m + N\boldsymbol{\sigma_1}$, then $(N, s)[a\rangle$, but also $(N, s)\,[b\rangle\,(N, s')$, for some $s' \in \mathbb{N}^P$, since $a^\bullet \cap {}^\bullet b = \emptyset$. Further we have $\neg(N, s')[a\rangle$, since otherwise $a \not\rightarrow_L b$. Therefore, $({}^\bullet b \setminus b^\bullet) \cap {}^\bullet a \neq \emptyset$. Let $p \in ({}^\bullet b \setminus b^\bullet) \cap {}^\bullet a$. Then, $s(p) = 1$, since if $s(p) > 1$, a would be enabled in (N, s'). Therefore, b is not enabled after firing $(\sigma; a)$. This is a contradiction, since now $\neg(N, s)[a; b\rangle$. □

Causal dependencies between transitions are used by many process discovery algorithms [5,7,15,21] and generally provide a good indication as to which transitions should be connected through places. Furthermore, extensive techniques are available to derive causal dependencies between transitions using heuristic approaches [7,15]. In order to find a place expressing a specific causal dependency, we extend the ILP presented in Definition 4.7.

Definition 4.13. (ILP for causal dependency) Let $L \in \mathcal{P}(T^*)$ be a log, let A and A' be the matrices defined in Definition 3.1 and let ILP_L be the corresponding ILP. Furthermore, let $t_1, t_2 \in T$ and assume $t_1 \rightarrow_L t_2$. We define the refined ILP, $ILP_{(L,t_1 \rightarrow t_2)}$ as ILP_L, with two extra bounds specifying that:

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

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

4.5 Net types

So far, we presented two algorithms for constructing a Petri net able to replay a log, using an ILP formulation. The two algorithms presented are generic and can easily be extended to different log types or to different net classes. In this subsection, we present possible extensions. For all of these extensions, we briefly sketch how they affect the ILP, and what the result is in terms of computational complexity.

Workflow nets Workflow nets [3] are a special class of Petri nets with a single marked input place and a single output place which are often used for modelling business processes. To search for workflow nets using our ILP approach, we do not allow for places to be marked, unless they have no incoming arcs. In terms of the ILP, this simply translates into saying that $c = 0$ when searching a place for a causal dependency and to separately search for initial places for each transition t not expressing a causal dependency, but with $c = 1$ and $\mathbf{1}^\mathsf{T} \cdot \boldsymbol{x} = 0$.

Figure 2 shows a Petri net constructed from the log of Table 1. This Petri net is *almost* what we consider to be a workflow net, i.e. it has a clear initial marking and the initially marked place does not have incoming arcs. When replaying the log of Table 1 in this net, it is obvious that the net is empty (i.e. no places contain tokens) after completion of each case, whereas workflow nets should contain a clearly marked final place when a case is complete. This property can also be expressed in terms of constraints, by demanding that the Petri net should have an empty marking after the completion of a case (in most cases, it is then rather easy to extend such a net to a net with a single output place).

Empty after case completion Another property which is desirable in process discovery, is the ability to identify the final marking. Using the ILP formulation

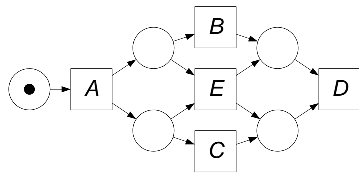

Fig. 2. Workflow net (without output place).

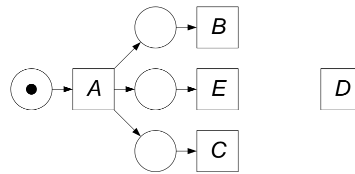

Fig. 3. Marked graph.

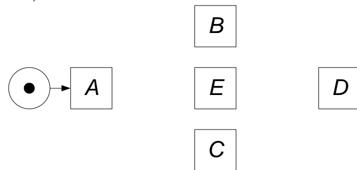

Fig. 4. Marked graph, empty after case completion.

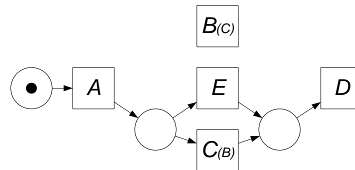

Fig. 5. State machine.

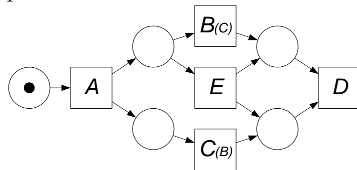

Fig. 6. Free choice.

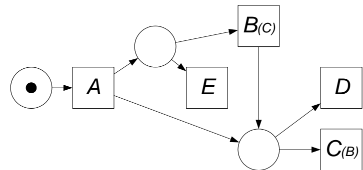

Fig. 7. Free choice, empty after case completion.

we presented in this paper, this can easily be achieved, by adding constraints ensuring that for all cases in the log which are no prefix of any other case in the log (or maximum cases), the net is empty when all transitions are fired. These constraints have the form $c + \boldsymbol{\sigma}^\mathsf{T} \cdot \boldsymbol{x} - \boldsymbol{a}^\mathsf{T} \cdot \boldsymbol{y} = 0$, where $\boldsymbol{\sigma}$ is the Parikh vector corresponding to a maximum case σ.

The requirement that the net has to be empty after case completion sometimes leads to a reduction of the number of feasible places. Consider for example a class of Petri nets called "marked graphs".

Marked Graphs In a marked graph [19], places have at most one input place and one output place, i.e. $\mathbf{1}^\mathsf{T} \cdot x \leq 1$ and $\mathbf{1}^\mathsf{T} \cdot \boldsymbol{y} \leq 1$. The influence of these constraints on the computation time is again negligible, however, these constraints do introduce a problem, since it is no longer possible to maximize the number of output arcs of a place (as it is at most 1). However, the procedure will find as many places with single input single output as possible.

In Figure 3, a marked graph is presented that was constructed from the log of Table 1. This net is clearly capable of replaying the log. However, after completion of a case, tokens remain in the net, either in the place before "E" or in the places before "B" and "C". When looking for a marked graph that is empty after case completion, the result is the net of Figure 4.

State machines State machines [19] are the counterpart of Marked graphs, i.e. transitions have at most one input place and one output place. It is easy to see that this cannot be captured by an extension of the ILP directly. The property is not dealing with a single solution of the ILP (i.e. a single place), but it is dealing with the collection of all places found.

Nonetheless, our implementation in ProM [4], which we present in Section 5 does contain a naive algorithm for generating state machines. The algorithm implemented in ProM proceeds as follows. First, the ILP is constructed and solved, thus yielding a place p to be added to the Petri net. Then, this place is added and from that point on, for all transitions $t \in {}^\bullet p$, we say that $\boldsymbol{x}(t) = 0$ and for all transitions $t \in p^\bullet$, we say that $\boldsymbol{y}(t) = 0$. Currently, the order in which places are found is undeterministic: the first place satisfying the conditions is chosen, and from that moment on no other places are connected to the transitions in its pre- and post-set.

Figure 5 shows a possible state machine that can be constructed using the log of Table 1. Note that transitions "B" and "C" are symmetrical, i.e. the figure actually shows 2 possible state machines, of which one will be provided by the implementation.

Free-Choice nets Similar to state machines, free choice nets [14] impose restrictions on the net as a whole, rather than on a single place. A Petri net is called free choice, if for all of its transitions t_1, t_2 holds that if there exists a place $p \in {}^\bullet t_1 \cap {}^\bullet t_2$ then ${}^\bullet t_1 = {}^\bullet t_2$.

Our implementation allows for the construction of free-choice nets and the algorithm used works as follows. First, all causal dependencies are placed on a stack. Then, for the first dependency on the stack, the ILP is solved, thus yielding a region $(\boldsymbol{x_0}, \boldsymbol{y_0}, c_0)$ with $\mathbf{1}^\mathsf{T} \cdot \boldsymbol{y_0} > 1$ corresponding to a place p with multiple outgoing arcs to be added to the Petri net. Then, this place is added and from that point on, constraints are added, saying for all $t_1, t_2 \in T$ with $\boldsymbol{y_0}(t_1) = \boldsymbol{y_0}(t_2) = 1$ holds that $\boldsymbol{y}(t_1) = \boldsymbol{y}(t_2)$, i.e all outgoing edges of the place added to the Petri net appear together, or none of them appears. If after this a place p_1 is found with even more outgoing edges than p, then p_1 is added to the Petri net, p is removed, the constraints are updated and the causal dependencies expressed by p, but not by p_1 are placed back on the stack. This procedure is repeated until the stack is empty.

It is easy to see that the algorithm presented above indeed terminates, i.e. places added to a Petri net that call for the removal of existing places always have more outgoing arcs than the removed places. Since the number of outgoing arcs is limited by the number of transitions, there is an upper bound to the number of removals and hence to the number of constraints placed back on the stack. The algorithm does however put a strain on the computation time, since each causal dependency is investigated as most as often as the number of transitions in the log, and hence instead of solving the ILP $|T|^2$ times, it might be solved $|T|^3$ times. However, since the added constraints have a specific form, solving the ILP gets quicker with each iteration (due to reduced complexity of the Branch-and-Bound part) [20].

Figure 6 shows a possible free-choice net that can be constructed using the log of Table 1. Note that transitions "B" and "C" are again symmetrical. Furthermore, the only difference between this net and the net of Figure 2 is that there is no arc from the place between "A" and "C" to "E". Adding this arc would violate the free-choice property. The fact that this arc is not there however does violate the property that the net is empty after case completion. Figure 7 shows a free-choice net that is empty after case completion. However, this net is no longer a so-called *elementary net*.

Pure nets Before introducing elementary nets, we first define *pure* nets [18], since elementary nets are pure. In a pure net, no self-loops occur. By adding a constraint $\boldsymbol{x}(t) + \boldsymbol{y}(t) \leq 1$, for each transition $t \in T$, each transition either consumes or produces tokens in a place, but not both at the same time. This procedure slightly increases the size of the ILP problem (with as many constraints as transitions found in the log), thus resulting in a slight increase in computation time. However, since most logs have far more prefixes than actual transitions, the overall effect is negligible.

Elementary nets Elementary Petri nets [17] are nets in which transitions can only fire when their output places are empty. This can easily be worked into the ILP, as shown in [17]. Two sets of constraints are required. First, self-loops are explicitly forbidden since elementary nets are pure and then, by adding the

constraints $c + \mathbf{1}^\mathsf{T} \cdot A \cdot \boldsymbol{x} - \mathbf{1}^\mathsf{T} \cdot A \cdot \boldsymbol{y} \leq 1$ it is ensured that after firing a transition each of its output places should contain at most one token. State machines, marked graphs and free-choice nets can all be made elementary this way. When requiring an elementary net however, the problem size doubles (there are twice as many constraints) and since the execution time is exponential in the size of the problem, the worst-case execution time is squared.

In this section, we presented a way of constructing a Petri net from an execution log using an ILP formulation. We presented a large number of net types and extensions to get a Petri net satisfying criteria set by a user. The figures on page 13 nicely show that with different sets of constraints, different models can be produced.

Although we used a toy example to illustrate the different concepts, we introduce our implementation embedded in the process discovery framework ProM, which is capable of constructing nets for logs with thousands of cases referring to dozens of transitions.

5 Implementation in ProM

The (Pro)cess (M)ining framework *ProM* [4] has been developed as a completely plugable environment for process discovery and related topics. It can be extended by simply adding plug-ins, and currently, more than 200 plug-ins have been added. The ProM framework can be downloaded from www.processmining.org.

In the context of this paper, the "Parikh language-based region miner" was developed, that implements the algorithms presented in Section 4. The Petri nets on Page 13 were all constructed using our plugin. For solving ILP problems, an open-source solver LpSolve [1] is used. Although experiments with CPlex [2] have been conducted, we feel that the use of the open-source alternative is essential: it allows for distribution of ProM, as well as reproducibility of the results presented in this paper.

Note that the plugin is also capable of dealing with partially ordered logs, i.e. logs where all cases are represented by partial orders on its events. The construction of the ILP in that case is largely the same as for the totally ordered cases as presented in [11, 17] and as included in VIPtool [10].

5.1 Numerical analysis

Using the implementation in ProM, we performed some performance analysis on our approach. We tested our algorithm on a collection of logs with varying numbers of transitions and varying numbers of cases, using the default settings of our plugin, which include the constraints for elementary nets and empty nets after case completion. Furthermore, we used logs that are distributed with the release of ProM and causal dependencies are used for guiding the solver.

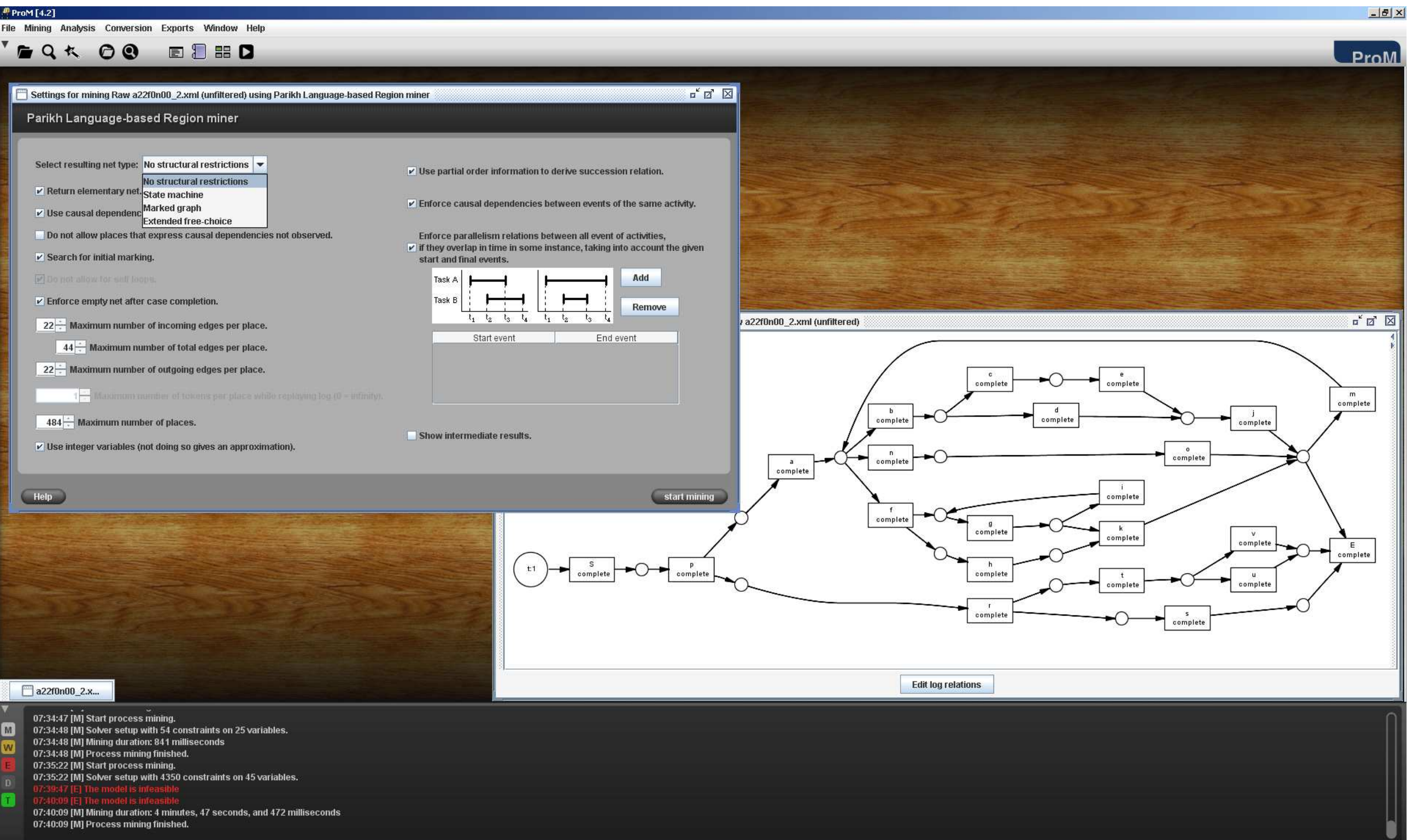

Fig. 8. ProM.

Table 2. Numerical analysis results.

log	# transitions	# variables	# cases	# events	# constraints	time (hh:mm:ss.sss)
a12f0n00_1	12	25	200	618	54	0.406
a12f0n00_1			600	1848	54	0.922
a12f0n00_3			1000	3077	54	1.120
a12f0n00_4			1400	4333	54	1.201
a12f0n00_5			1800	5573	54	1.234
a22f0n00_1	22	45	100	1833	1894	1:40.063
a22f0n00_2			300	5698	4350	5:07.344
a22f0n00_3			500	9463	5863	7:50.875
a22f0n00_4			700	13215	7238	10:24.219
a22f0n00_5			900	16952	8405	12:29.313
a32f0n00_1	32	65	100	2549	3352	32:14.047
a32f0n00_2			300	7657	7779	1:06:24.735
a32f0n00_3			500	12717	10927	1:46:34:469
a32f0n00_4			700	17977	13680	2:43:40.641
a32f0n00_5			900	23195	15978	2:54:01.765

Note that solving an ILP problem consists of two stages. The first stage uses the Simplex algorithm, which is worst-case exponential, but generally outperforms polynomial algorithms and the second phase which is an exponential search.

The results presented in Table 2 show that adding cases to a log referring to a given number of transitions increases the necessary calculation time in a seemingly sub-linear fashion[2], however this might not be the case in general. Figure 8 shows a screenshot of ProM, showing the Petri nets discovered for the log "a22" with 300 cases in the bottom right corner. The settings dialog for our plugin is shown in the top-left corner.

Since for the log "a12" the number of constraints remains constant for different log sizes, the increase in processing time is only due to the overhead of calculating the causal dependencies, which is polynomial in the size of the log. For the other logs however, the increase in processing time is due to the fact that adding constraints influences the first phase of solving an ILP problem, but not the second phase which is far more complex. Furthermore, our experiments so-far show an almost linear dependency between the number of cases in the log and the execution time, although this might not be the case in general.

On the other hand, when increasing the number of transitions (i.e. the number of variables) instead of the number of case, the branch-and-bound phase is heavily influenced. Hence the computation time increases exponentially, since the branch-and-bound algorithm is worst-case exponential. However, these results show that the execution time is still feasible, i.e. results for a large log can be generated overnight.

6 Conclusion and Future work

In this paper we presented a new method for process discovery using integer linear programming (ILP). The main idea is that places restrict the possible

[2] These calculations were performed on a 3 GHz Pentium 4, using ProM 4.2 and LpSolve 5.5.0.10 running Java 1.5. The memory consumption never exceeded 256MB.

firing sequences of a Petri net. Hence we search for as many places as possible, such that the resulting Petri net is consistent with the log, i.e. such that the Petri net is able to replay the log.

The well-known theory of regions solves a similar problem, but for finite languages. Finite languages can be considered as prefix closures of a log and the theory of regions tries to synthesize a Petri net which can reproduce the language as precisely as possible. In [11, 17] this idea is elaborated and the problem is formalized using a linear inequation system. However, the Petri nets synthesized using the approach of [11, 17] scale in the number of events in the log, which is undesirable in process discovery.

In this paper, we build on the methods proposed in [11, 17]. First of all we have defined an optimality criterion transforming the inequation system into an ILP. This ILP is than solved under different conditions, such that the places of a Petri net capable of replaying the log are constructed.

The optimality criterion we defined is such that it guarantees that more expressive places are found first, i.e. places with less input arcs and more output arcs are favored. Furthermore, using the causality relation that is used in the alpha algorithm [7], we are able to specifically target the search to places expressing this relation. This causality relation is generally accepted in the field of process discovery and, under the assumption the log is complete, is shown to directly relate to places. Furthermore, the size of the constructed Petri net is shown to be independent on the number of events in the log, which makes this approach applicable in more practical situations.

Using additional constraints, we can enforce structural net properties of the discovered Petri net, such as the freedom of choice. It is clear that not all these constraints for structural properties lead to feasible solutions, but nonetheless, we always find a Petri net that is consistent with the log. For all our constraints we provide lemmas motivating these constraints.

The numerical quality of our approach is promising: we can discover nets with about 25 transitions and a log with about 1000 cases in about 15 minutes on a standard desktop PC. Moreover, while the execution time of our method appears to scale sub-linear in the size of the log, although this needs to be validated more thoroughly.

Since each place can be discovered in isolation, it seems to be easy to parallelize the algorithm and to use grid computing techniques to speed up the computation time. Other open questions concern the expression of more structural and behavioral properties into linear constraints and the use of this algorithms in a post-processing phase of other algorithms.

References

1. LP Solve Reference guide. http://lpsolve.sourceforge.net/5.5/.
2. ILOG CPLEX, ILOG, Inc., 2006. http://www.ilog.com/products/cplex/.
3. W.M.P. van der Aalst. Verification of Workflow Nets. In *ICATPN 1997*, pages 407–426, London, UK, 1997. Springer-Verlag.

4. W.M.P. van der Aalst, B.F. van Dongen, C.W. Günther, R.S. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H.M.W. Verbeek, and A.J.M.M. Weijters. ProM 4.0: Comprehensive Support for Real Process Analysis. In J. Kleijn and A. Yakovlev, editors, *Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*, volume 4546 of *lncs*, pages 484–494. springer, 2007.
5. W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.
6. W.M.P. van der Aalst, V. Rubin, B.F. van Dongen, E. Kindler, and C.W. Günther. Process Mining: A Two-Step Approach using Transition Systems and Regions., 2007.
7. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
8. E. Badouel, L. Bernardinello, and Ph. Darondeau. Polynomial Algorithms for the Synthesis of Bounded Nets. In *TAPSOFT*, pages 364–378, 1995.
9. E. Badouel and Ph. Darondeau. Theory of regions. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *lncs*, pages 529–586. springer, 1998.
10. R. Bergenthum, J. Desel, G. Juhás, and R. Lorenz. Can I Execute My Scenario in Your Net? VipTool Tells You! In *Petri Nets and Other Models of Concurrency - ICATPN 2006*, 2006.
11. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Process mining based on regions of languages. In G. Alonso, P. Dadam, and M. Rosemann, editors, *BPM*, volume 4714 of *Lecture Notes in Computer Science*, pages 375–383. Springer, 2007.
12. J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri nets from Finite Transition Systems. *IEEE Transactions on Computers*, 47(8):859–882, AUG 1998.
13. Ph. Darondeau. Deriving Unbounded Petri Nets from Formal Languages. In *CONCUR '98: Proceedings of the 9th International Conference on Concurrency Theory*, pages 533–548, London, UK, 1998. Springer-Verlag.
14. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.
15. B.F. van Dongen. *Process Mining and Verification*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2007.
16. A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures - Part 1 and Part 2. *Acta Informatica*, 27(4):315–368, 1989.
17. R. Lorenz and R. Juhás. How to Synthesize Nets from Languages - a Survey. In *Proceedings of the Wintersimulation Conference (WSC) 2007*, 2007.
18. J.L. Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, Englewood Cliffs, 1981.
19. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *Monographs in Theoretical Computer Science: An EATCS Series*. Springer-Verlag, Berlin, 1985.
20. A. Schrijver. *Theory of Linear and Integer programming*. Wiley-Interscience, 1986.
21. A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.