An ILASP-Based Approach to Repair Petri Nets

Francesco Chiariello[1][0000-0001-7855-7480], Antonio Ielo[2][0009-0006-9644-7975], and Alice Tarzariol[3][0000-0001-6586-3649]

[1] ANITI, IRIT, University of Toulouse, France
francesco.chiariello@irit.fr
[2] University of Calabria, Italy
antonio.ielo@unical.it
[3] University of Klagenfurt, Austria
alice.tarzariol@aau.at

Abstract. Petri nets are a class of models of computation used to compactly represent discrete event systems. Among many application domains, they have now become the most prominent formalism to express process models in Process Mining, thanks to their formal semantics that enables automated analysis techniques. In this context, *model repair* is the task of aligning a process model with actual executions of the process. Current solutions to model repair do not allow for embedding domain knowledge, providing guarantees of rigor, and enforcing structural requirements at the same time. In this paper, we fill this gap by proposing an approach based on the Inductive Logic Programming system ILASP. We then implement our approach and perform an experimental evaluation, showing both its expressiveness and feasibility.

Keywords: Petri nets · Process Mining · Model Repair · Answer Set Programming · Inductive Logic Programming.

1 Introduction

Process Mining (PM) [3] is an interdisciplinary field at the intersection of Business Process Management (BPM) [29] and Data Mining that aims at getting insight into operational processes by analyzing event logs as recorded by enterprise information systems. An operational process, or simply a *process*, expresses the relationships among the activities an organization performs to achieve a goal, such as delivering a particular service or product. Several formalisms have been proposed to model and reason about processes, including Linear Temporal Logic on process traces [19, 12] and Petri nets [2, 4], the *de facto* standard model in BPM. An *event log*, i.e., the collection of activities the organization executes while enacting the process, can then be analyzed to perform various PM tasks. In particular, *Model Repair* is the task of revising a process model to make it conformant to the log [17]. This is of particular interest in heavily regulated environments, where handwritten process models need to be updated to reflect the actual behavior of the organization while, at the same time, preserving as

much information as possible from the original model, usually containing relevant domain knowledge. Multiple solutions exist to repair Petri nets [18, 26, 7]. However, none of them allow the user to easily specify the edit operations, the structural properties of the net [1], or to exploit domain knowledge [27], e.g., by specifying portions of the net that should not be involved with the edits. Besides, those approaches do not guarantee full conformance to the log. In this paper, we work towards overcoming these limitations, by proposing a fully declarative, user-customizable framework for Petri net model repair based on the Inductive Logic Programming system ILASP. Our experiments show the proposed approach is effective on medium-sized Petri nets where domain knowledge about repairs is available.

Related works. The model repair problem was first proposed in [25]. This work is similar to ours in that they both consider Petri nets as the language to express process models and use Inductive Logic Programming (ILP; [14]) as a solution approach. With respect to this work, our approach is more general in that it does not *require* — but supports — negative examples, and it allows users to define arbitrary edit operations. The first procedural approach to model repair is proposed in [18]. Such an approach is based on adding subprocesses to the original net. The approach computes non-fitting subtraces in a place and then applies standard discovery algorithms to determine the subprocess accepting those subtraces. The subprocesses, once added to the model, make the whole trace accepted. Differently from our approach, their approach does not allow the removal of arcs or nodes from the input model, thus significantly increasing the size of the Petri net and decreasing its readability. In [26], a set of edit operations is defined, each with a corresponding cost, and the model repair problem is formalized as a search for edits that maximizes the *fitness score* within a total cost bound. However, the only edits allowed are insertion and skipping of activities, rather than structural changes of the net. Related works have already investigated the application of ILP and Answer Set Programming (ASP; [24]) to Process Mining. Besides being used for model repair [25], ILP has also been applied in other Process Mining tasks. It was first put forward in [21] as a language to represent processes as sets of integrity constraints. Then, it was used in [9] to learn declarative models. These works use logic programming constraints directly as a process model rather than as a language to represent the process models. This approach has been followed by recent works about ASP applications to Process Mining [11, 10, 13].

2 Background

This section recaps basic definitions that will be relevant for the rest of the paper, namely, Petri nets, Answer Set Programming, and Inductive Logic Programming.

2.1 Event Logs and Petri nets

Each execution of a process generates a *trace*, which is modeled as a finite sequence $\pi \in \Sigma^*$, where Σ is a finite set of symbols that model the process activi-

ties. For our purposes, an *event log* is a multiset of traces, representing examples of process behavior. An *(accepting) Petri net* $\mathcal{M} = (P, T, W, m_0, m_f, \lambda)$ is defined by a weighted bipartite graph over two sets P and T, called respectively *places* and *transitions*. The graph's directed edges E are implicitly defined by the function W that assigns to each arc $e \in (P \times T) \cup (T \times P)$ a weight $w \in \mathbb{N}_{>0}$ if $e \in E$ and 0 if $e \notin E$. For each $x \in (P \cup T)$, its sets of incoming and outgoing edges are respectively denoted as ${}^{\bullet}x$ (called *pre-set*) and $x^{\bullet}$ (*post-set*). A *marking* is a function $m : P \mapsto \mathbb{N}$ that assigns to each place a certain number of *tokens*. The markings m_0 and m_f of the net are referred to as the *initial* and *final* marking, respectively. The *labeling function* $\lambda : T \to \Sigma$ associate to each transition $t \in T$ an activity $\lambda(t) \in \Sigma$.

We say that a transition t is *enabled* in a marking m if $m(p) \geq W(p, t)$ for all $p \in {}^{\bullet}t$. *Firing* t in m yields a new marking m' defined by $m'(p) = m(p) - W(p, t) + W(t, p)$ for each $p \in P$. When t fires, it consumes $W(p, t)$ tokens from each place $p \in {}^{\bullet}t$ and produces $W(t, p)$ new tokens in each place $p \in t^{\bullet}$. A sequence of transitions $t_1, \ldots, t_k$ is a *complete firing sequence* for a Petri net $\mathcal{M}$ if, for each $i \in [1..k]$, the transition t_i is enabled in m_{i-1}, and firing it yields m_i, where $m_k = m_f$. Applying λ to each transition of a complete firing sequence $t_1, \ldots, t_k$ induces a *trace* $\lambda(t_1), \ldots, \lambda(t_k)$. The *language* $\mathcal{L}(\mathcal{M})$ of a Petri net $\mathcal{M}$ is the set of its traces. Intuitively, transitions are associated with activities that occur within a process execution, while places implicitly model pre- and post-conditions of activities. The initial and final markings encode, respectively, the starting condition and the desired final state after the execution of the process.

2.2 Answer Set Programming

Answer Set Programming (ASP; [8]) is a declarative programming paradigm widely used to solve combinatorial search and optimization problems. This section briefly recaps the syntax and semantics of ASP. We assume that the readers are familiar with logic and refer them to [20, 24] for a more exhaustive overview.

Syntax. An ASP program Π is a finite set of *(normal) rules* r, defined as:

$$a_0 \leftarrow a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n.$$

where each a_i for $i \in [0..n]$ is an atom, and *not* stands for *default negation*. An *atom* is expressed as $p(\bar{t})$, where p is a predicate, and $\bar{t}$ is a (possibly empty) vector of *term*, each of which is either a variable or a constant. $H(r) = \{a_0\}$ is known as *head* of the rule r, while the *body* of r is partitioned into the positive or negative body atoms, respectively defined as $B^+(r) = \{a_1, \ldots, a_m\}$ and $B^-(r) = \{a_{m+1}, \ldots, a_n\}$. The rule r is called a *fact* if $B^+(r) \cup B^-(r) = \emptyset$, and a *constraint* if $H(r) = \{\bot\}$, where $\bot$ is a predicate without terms representing the constant *false*. As syntactic sugar, we can write a *choice rule* c of the form:

$$b\{a_1; \ldots; a_k\}u \leftarrow a_{k+1}, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n.$$

where each a_i, for $i \in [0, .., n]$, is an atom, and b and u are non-negative integers. The *head* of c is $H(c) = \{a_1, \ldots, a_k\}$.

Semantics. The semantics of Π is defined in terms of its equivalent *propositional instantiation* $G(\Pi)$. To compute it, we replace each rule $r \in \Pi$ with all its possible instantiation obtained by substituting the variables in r with constants occurring in Π. An *interpretation* $\mathcal{I}$ is a set of (*true*) propositional atoms occurring in $G(\Pi)$ that does not contain $\bot$. An interpretation $\mathcal{I}$ *satisfies* a normal rule $r \in G(\Pi)$ if $B^+(r) \subseteq \mathcal{I}$ and $B^-(r) \cap \mathcal{I} = \emptyset$ imply $H(r) \subseteq \mathcal{I}$. Similarly, $\mathcal{I}$ *satisfies* a choice rule $c \in G(\Pi)$ if $B^+(c) \subseteq \mathcal{I}$ and $B^-(c) \cap \mathcal{I} = \emptyset$ imply $b \leq |H(c) \cap \mathcal{I}| \leq u$. The interpretation $\mathcal{I}$ is a *model* of Π if it satisfies all rules $r \in G(\Pi)$ and all choice rules $c \in G(\Pi)$, and $\mathcal{I}$ is *stable* if it is a subset-minimal model of the reduct $\{H(r) \leftarrow B^+(r) \mid r \in \Pi, B^-(r) \cap \mathcal{I} = \emptyset\} \cup \{a_0 \leftarrow B^+(c) \mid c \in G(\Pi), B^-(c) \cap \mathcal{I} = \emptyset, a_0 \in H(c) \cap \mathcal{I}\}$. The set of all stable models of Π, also called *answer sets*, is denoted by $AS(\Pi)$.

2.3 Learning from Answer Sets

Inductive Logic Programming (ILP) is a symbolic machine learning technique that uses logic programs as inductive bias [14]. ILASP is a state-of-the-art ILP system to learn ASP programs [22]. A *learning from answer sets* (LAS) task is a tuple $\mathcal{T} = (H, B, E^+, E^-)$, where H is the *hypothesis space*, defining a set of candidate rules, B is an ASP program called *background knowledge*, while E^+ and E^- are set of *positive* and *negative examples*, respectively. Each example is denoted with a triple (Inc, Exc, C), where Inc and Exc are sets of atoms, called *inclusion* and *exclusion set*, respectively, and C is an ASP program, called *context*. We say that an interpretation $\mathcal{I}$ is an *accepting answer set* of an example (Inc, Exc, C) with respect to B if $\mathcal{I} \in AS(B \cup C)$ such that $Inc \subseteq \mathcal{I}$ and $Exc \cap \mathcal{I} = \emptyset$. Given a LAS task $\mathcal{T}$, ILASP searches for an *inductive solution* $h \subseteq H$ such that (i) for each positive example $e \in E^+$, there is some accepting answer set of e with respect to $B \cup h$, and (ii) for any negative example $e \in E^-$, there is no accepting answer set of e with respect to $B \cup h$. Together with $\mathcal{T}$, ILASP allows the definition of a scoring function σ to be used as optimization criteria, which by default is the length of the solution. ILASP returns an inductive solution h that is optimal wrt σ, i.e., there is no other $h' \subseteq H$ such that $\sigma(h') < \sigma(h)$.

3 Model Repair in ILASP

In the context of process mining, the *Model Repair* problem consists of modifying a process model $\mathcal{M}$ to make it conform to an event log L [17]. Especially when $\mathcal{M}$ originates from domain experts' modeling, it is desirable to repair it while preserving as much structure as possible, to potentially retain its interpretability and valuable information. In this section, we illustrate how to set the elements of a LAS task (namely, hypothesis space, background knowledge, and examples) to tackle the model repair of a Petri net. The learned solution represents edit operations that, applied to $\mathcal{M}$, make it satisfy the structural requirements and accept all the traces in L.

Given a Petri net $\mathcal{M} = (P, T, W, m_0, m_f, \lambda)$, we denote by $[\mathcal{M}]$ its encoding into ASP containing the following facts:

- $place(p)$ for each $p \in P$;
- $trans(t, l)$ for each $t \in T$, where $\lambda(t) = l$;
- $ptarc(p, t, w)$ for each $p \in P$ and $t \in T$, where $W(p, t) = w$ and $w > 0$;
- $tparc(t, p, w)$ for each $p \in P$ and $t \in T$, where $W(t, p) = w$ and $w > 0$;
- $initial_marking(p, n)$ where $m_0(p) = n$ with $n > 0$;
- $final_marking(p, n)$ where $m_f(p) = n$ with $n > 0$.

Throughout the section, we will refer to the Petri net $\mathcal{M}$ as the *original model* and define the ASP program $\Pi_{\mathcal{M}} = \{original(x) : x \in [\mathcal{M}]\}$.

3.1 Hypothesis Space

The hypothesis space of our ILP task contains *edit operations* that are allowed on the original Petri net $\mathcal{M}$. The formal semantics of edit operations is provided in terms of ASP rules, contained in a program called Π_H. We define the hypothesis space explicitly through Π_H as this setting yields a flexible framework where users can customize the model repair actions in a declarative fashion. Given an original Petri net $\mathcal{M}$, $AS(\Pi_H \cup [\mathcal{M}])$ contains all the possible edit actions that ILASP considers for $\mathcal{M}$. We assume that each input action is an atom of the form $add(x)$ or $remove(x)$, where x can be any fact defined in $\mathcal{M}$. In the following, we provide an example of a possible definition of Π_H involving only edits to the arcs of the bipartite graph:

```
1  remove(tparc(T,P,W)) :- place(P), trans(T,_), tparc(T,P,W).
2  remove(ptarc(P,T,W)) :- place(P), trans(T,_), ptarc(P,T,W).
3  add(tparc(T,P,1)) :- place(P), trans(T,_), not tparc(T,P,_).
4  add(ptarc(P,T,1)) :- place(P), trans(T,_), not ptarc(P,T,_).
5  abducible(add(A),1) :- add(A).
6  abducible(add(A),C) :- add(A,C).
7  abducible(remove(A),1) :- remove(A).
8  abducible(remove(A),C) :- remove(A,C).
9  { head(A): abducible(A,C) } = 1.
10 #show. #show (C,A) : head(A), abducible(A,C).
```

The first four lines define the possible edit actions considered in this example: lines 1-2 define a set of atoms $remove(x)$, for each x matching the signature *tparc* or *ptarc* in $[\mathcal{M}]$; while lines 3-4 define $add(x)$, for each x matching the signature *tparc* or *ptarc* involving places and transitions in $[\mathcal{M}]$ without edges connecting them (in this example, the new arcs have a default weight equal to 1). Lines 5-8 define the possible abducible atoms, generating the search space through the choice rule at line 9. Lastly, line 10 specifies to return each answer set as a pair `(C,A)` where `C` represents the cost of including each action in the learned hypothesis, i.e., $\sigma($`A`$)=$`C`. Lines 5 and 7 set `C` to 1 if no cost is specified (as is the case in our example); however, lines 6 and 8 allow us to recognize different costs if a second term is used with *add* or *remove*. Our framework enumerates all the solutions in $AS(\Pi_H \cup [\mathcal{M}])$ and transforms each pair `(C,A)` to reflect the ILASP syntax of an element of the search space, namely: `C ~A.`, storing them in a temporary file H.

The above program shows edit actions involving only the arcs of $\mathcal{M}$; however, any element can be added or removed from the search space by the addition of similar rules. For example, in our experiments in Section 4.1, we add the rule `remove(place(P)) :- place(P)` to also allow actions that remove places from $\mathcal{M}$. It is important to keep in mind that increasing the search space, may lead to a considerable increase in the time required to solve the ILP problem. Consequently, it is desirable to limit the space of possible modification actions as much as possible. If the users desire to find a set of edit actions that avoid editing part of the original Petri net (for instance, because they know that those elements are correct, or they want to keep the exact meaning for that part of the model) they can provide facts of the form $frozen(x)$ to prevent any kind of edit involving the place or transition x. Then, it is sufficient to add the literals `not frozen(P)` and `not frozen(T)` in the body of any rule defining *remove* and *add*. We evaluate these effects in our experiments in Section 4.2.

3.2 Background Knowledge

The background knowledge B of our learning task is split into three parts: $\Pi_{\mathcal{M}}$, Π_{edit}, which produces a new Petri net by applying the edit operations to $\mathcal{M}$, and lastly $\Pi_{semantics}$, which represent the ASP program modeling the Petri net semantics. The logic program Π_{edit} contains the following rules:

```
1 valid(E) :- original(E), not remove(E).
2 valid(E) :- add(E).
3 place(P)   :- valid(place(P)).
4 trans(T,L) :- valid(trans(T,L)).
5 ptarc(P,T,W) :- valid(ptarc(P,T,W)).
6 tparc(T,P,W) :- valid(tparc(T,P,W)).
7 initial_marking(T,W) :- valid(initial_marking(T,W)).
8 final_marking(T,W) :- valid(final_marking(T,W)).
```

Lines 1-2 assume as facts the edit operations $add(e)$ and $remove(e)$ selected by ILASP as the current inductive hypothesis and define how they affect the original model, producing in lines 3-8 a revised Petri net, $\mathcal{M}'$.

The ASP program $\Pi_{semantics}$ is an adaption of the one introduced in [6]. It takes in input the original model $\Pi_{\mathcal{M}}$ and Π_{edit}, together with a trace modeled by a set of facts $trace(i, l)$, stating that at the time i, the label l was observed. Then, it verifies whether the considered trace represents a complete firing sequence with respect to $\mathcal{M}'$. The facts over the predicate *trace* are defined in the context of ILASP examples. Similarly to the encoding in [6], $\Pi_{semantics}$ defines the atoms $time(i)$ with i ranging from 0 to k (where k is the length of the trace), establishing the horizon time for computing the marking. The atoms $holds(p, q, i)$ entail that after firing the i-th transition, there are q tokens on place p. The atoms $enabled(t, i)$ are true if the transition t is enabled at time i, while the atoms $fires(t, i)$ are defined non-deterministically through a choice rule, stating that the transition t is fired at time i. $\Pi_{semantics}$ differs from encoding in [6] with respect to the firing condition, as we consider the interleaving semantics:

```
1 0 { fires(T,TS) } 1 :-
2   trace(TS,A), trans(T,A), enabled(T,TS), time(TS), last(L), TS <= L.
3 :- not fires(_,TS), time(TS), last(L), TS <=L.
4 :- fires(T1,TS), fires(T2,TS), T1 > T2, time(TS).
5 :- last(L), final_marking(F,T), not holds(F, T, L+1).
6 :- last(L), final_marking(F,T), place(P), P != F, holds(P,C,L+1), C > 0.
```

The rule in lines 1-2 chooses the atoms $fires(t, i)$ for each time point $i \in [1..k]$, where k is the length of the trace, encoded by the atom $last(k)$. Considering the constraints in lines 3-4, for each i, there is exactly one atom $fires(t, i)$ defining the fired transition, selected among those enabled at time i and with a label matching the activity specified by the trace. Lines 5-6 define the condition to recognize the current trace as a complete firing sequence; namely, at the last step, the marking produced by a firing sequence for the input trace matches m_f.

3.3 Positive and Negative Examples

The positive and negative examples of our LAS task derive from two elements: the event log L and the structural requirements. The former can be split into two sets, L^+ and L^- containing respectively traces that should or should not characterize the considered process. We allow this flexibility since it has recently been observed that negative examples are a valuable source of information, and thus being increasingly adopted in industry [5, 28]. However, our setting allows performing model repair even when L^- is empty. Each trace $\pi = a_0, \ldots, a_k \in L^+$ (resp. L^-) is encoded as a context-dependent positive example (resp. negative example), with empty inclusions and exclusions, while the context is the set of ASP facts $trace(i, a_i)$ stating that the i-th event in the trace is the activity a_i. In this way, we use ILASP examples to encode input traces.

Example 1. Consider two traces $acef \in L^+$ and $acf \in L^-$; their examples are:

```
#pos(id1, {}, {}, { trace(0,a). trace(1,c). trace(2,e). trace(3,f).} ).
#neg(id2, {}, {}, { trace(0,a). trace(1,c). trace(2,f).} ).
```

Besides conformance on L, we also exploited negative examples to enforce structural properties to the revised model by defining in their context the ASP rules containing constraints that are triggered when a desired condition is not met.

Example 2. Consider the following example, forcing to obtain a Petri net where each node can be reached in the execution of a complete firing sequence.

```
1 #neg(id3, {}, {}, {
2   node(P) :- place(P).
3   node(T) :- trans(T,_).
4   edge(X,Y) :- ptarc(X,Y,_).
5   edge(X,Y) :- tparc(X,Y,_).
6   start(X) :- initial_marking(X,_).
7   end(X)   :- final_marking(X,_).
8   reach(X,X) :- start(X).
```

```
9     reach(X,X) :- end(X).
10    reach(X,Y) :- edge(X,Y).
11    reach(X,Z) :- reach(X,Y), edge(Y,Z).
12    :- node(X), start(S), not reach(S,X).
13    :- node(X), end(S), not reach(X,S).
14    :- edge(S,_), end(S).
15    :- edge(_,S), start(S). }).
```

The connected property is obtained with a reachability test. Lines 2-5 define the nodes and edges of the graph, considering places and transitions identically. Lines 6-7 set the places in the initial and final marking as possibly starting and ending nodes, respectively. Lines 8-13 define the classical reachability property, and lines 14-15 state that there should be no ingoing or outgoing edges from a place that is an initial or final marking, respectively.

3.4 Learned Hypothesis and Implementation Notes

To recap, our setting defines a LAS task (H, B, E^+, E^-), where the hypothesis space H is produced by the program Π_H, the background knowledge B consists of $\Pi_\mathcal{M} \cup \Pi_{edit} \cup \Pi_{semantics}$ and the set of positive examples E^+ is obtained from L^+, while the negative examples E^- are derived from L^-, as well as from the structural requirements. Then, ILASP will search for the optimal hypothesis $h \subseteq H$ where the logic program $\Pi_\mathcal{M} \cup \Pi_{edit} \cup h$ represents the revised Petri net $[\mathcal{M}']$ and for each trace defined in the context C of a positive example, $[\mathcal{M}'] \cup \Pi_{semantics} \cup C$ is satisfiable (namely, $C \in \mathcal{L}(\mathcal{M}')$), while for each negative example $[\mathcal{M}'] \cup \Pi_{semantics} \cup C$ is unsatisfiable (namely, $C \notin \mathcal{L}(\mathcal{M}')$). When C represents structural characteristics, then $\mathcal{M}'$ is conformant to them.

Our setting uses version 2i of ILASP since we observe the best performance for learning tasks that do not consider noisy examples (although this option is easily achievable by design [22]). During our experiments (Section 4), we observed a clear speed-up[4] in the learning time when only positive examples were used. For this reason, we revised the background knowledge and examples in our implementation to encode the equivalent learning tasks expressed only with positive examples. To do so, we add the atom `fail` in the head of each constraint of the background knowledge and in the context of the examples [23]. Then, we add the rule `ok :- not fail` in $\Pi_{semantics}$ and rewrite every example as follows: for each positive example, we add in the inclusion set the atom `ok`; while each negative example is transformed in a positive example, adding in the exclusion set the atom `ok`.

Example 3. Consider the two traces from Example 1. We can express them as positive examples as follows:

```
#pos(id1, {ok}, {}, { trace(0,a). trace(1,c). trace(2,e). trace(3,f).} ).
#pos(id2, {}, {ok}, { trace(0,a). trace(1,c). trace(2,f).} ).
```

[4] We conjecture this might be related to how negative examples are handled in ILASP.

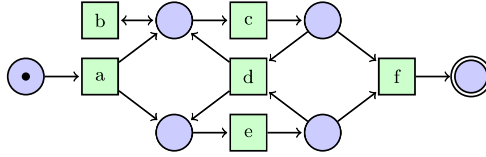


Fig. 1: A workflow net (from [1]) with a token in its initial place (we indicate the final place with double circle). Places are circles and transitions are squares with the label written in them. Each arc has a weight equal to 1.

4 Implementation and Experiments

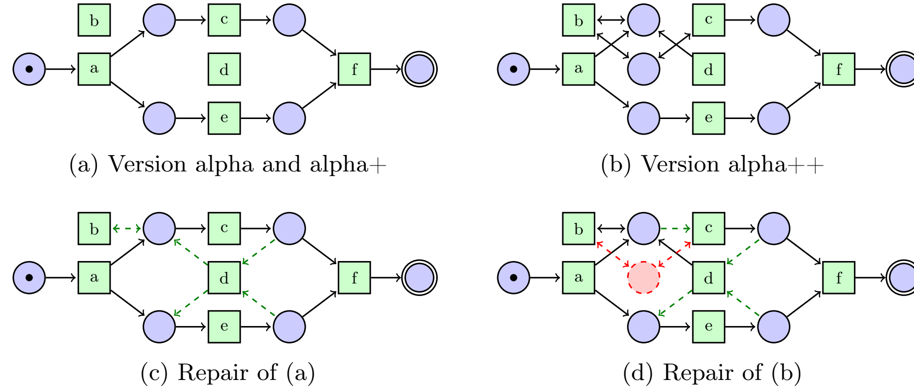
After defining our approach, we implemented and assessed its feasibility for tackling the model repair problem. Our code, settings, and detailed results from experiments are available online [5]. In the repository, we provide scripts to map the default representation formats for logs (i.e., XES) and Petri nets (i.e., PNML) into a LAS task, as well as helper scripts to define the hypothesis space from the file Π_H. The experiments consider a test case to evaluate the quality of the repairs found, as well as aggregate results for evaluating the performance of our approach with respect to the percentage of edits introduced on a synthetic model originating from real logs. We run the experiments on a MacBook Pro (Apple M3 Pro Chip with 18GB RAM on Sonoma 14.5), running ILASP version 4.4.0 and the ASP system clingo 5.7.1.

4.1 Repair Discovered Models

For our first experiment, we targeted a Petri net with structural requirements [1]. More precisely, we consider a *WorkFlow net* (WF-net), namely a Petri net where (i) there exists a place i such that ${}^\bullet i = \emptyset$, called the *initial place*; (ii) there exists a place o such that $o^\bullet = \emptyset$, called the *final place*; (iii) every place and every transition is located on a path from i to o. Here, the initial (resp. final) marking is intended to be the marking with exactly one token in the initial (resp. final) place and no token in the other places.

Starting from the WF-net in Fig 1, we generate a log containing 1000 traces by simulation. Their length ranges from 4 to 63 activities, with an average of 9. This yields 369 execution variants. Then, we use the *Alpha Miner* algorithm, as implemented in ProM [16], to discover a model from the log, obtaining the model in Fig. 2a with the version Alpha and Alpha+, and the model in Fig. 2b with Alpha++. This plugin quickly discovers Petri nets, however it is not possible to specify structural properties, nor there are guarantees on a complete coverage of the traces in the log. Indeed, none of the discovered models are WF-nets. The one in Figure 2a has two disconnected transitions — the ones labeled with b and d —, while in the one in Figure 2b the transition labeled with d is not on a path from the initial to the final place. Moreover, the language of the original model

[5] `www.github.com/ainnoot/ilp-pn-repair`



(a) Version alpha and alpha+ (b) Version alpha++

(c) Repair of (a) (d) Repair of (b)

Fig. 2: Models learned with Alpha Miner (a), (b) and their repair, respectively (c), (d)

in Fig 1, contains traces with multiple executions of activities c and e, while, the one of model in Fig 2a contains only traces with a single execution of these activities, and the language of Fig 2b is empty.

To test our approach, we consider the two models produced by Alpha Miner and repair them to be conformant to the log and satisfy the structural properties of a WF-net. More precisely, we transform each trace in the log as a positive example and add the negative[6] example introduced in Example 2 that characterize the structure of a WF-net. The hypothesis space contains edit actions that add arcs or remove arcs or places. In Figure 2c and Figure 2d, we can see that the repairs, depicted as dashed elements, successfully restore the original model.

4.2 Assess Performance

For our second experiment, we consider a process model extracted from the *Domestic Declaration* event log [15], collecting events about expenses approval & reimbursement process in an academic setting. From this event log, we learn a model using the *Inductive Miner* algorithm with the default parameters using the ProM toolkit [16]. We prefer it over the Alpha Miner for its ability to discover more accurate models from complex, real-world data. The model obtained is a WF-net with 16 places, 25 transitions (for a total of 41 nodes), and 52 edges. Subsequently, we edited the model by randomly adding or removing 20 arcs (which is equivalent to 40% of the model's edges) and applied our approach to find the repairs that restored the original model. The language of the model is finite, producing a total of 424 traces that were used as positive examples, and we consider a single negative example for the structural property of a WF-net (see Example 2). We tested the edits incrementally (ranging from 10% to 40% of the model's edges) to evaluate the overlap between the learned hypothesis and the original model, as well as the learning time required to obtain them. At the

[6] Because of performance, we transform it in a positive example; see Section 3.4.

	10% Edits			20% Edits			30% Edits			40% Edits		
	#s	Ovlp	Time	#s	Ovlp	Time	#s	Ovlp	Time	#s	Ovlp	Time
0%	10	94.0	0.885	10	94.0	15.870	9	88.7	42.204	7	82.9	53.771
4%	10	96.0	0.747	10	94.0	3.263	9	88.6	21.800	9	86.7	42.584
8%	10	96.0	0.554	10	96.0	1.681	10	92.4	49.469	10	91.0	25.151
12%	10	98.0	0.546	10	93.0	0.910	10	93.1	1.520	10	91.5	3.279

Table 1: Aggregate results for different percentages of edits on the edges (in the columns) and frozen nodes (in the rows). *#s* represents the number of runs finished within the timeout of 600 seconds, for a total of 10; *Ovlp* and *Time* represent, respectively, the *overlap* — percentage of facts the learned hypothesis that belongs to the original model — and learning time (considering only the runs that finish within the timeout).

same time, we assess the effect of removing some elements from the search space, by adding a number of facts with the predicate *frozen* in Π_H, ranging from 0% to 12% of the model's nodes. In this setting, including negative examples (through ILASP's `#neg` qualifier) caused timeouts over all instances. Thus, we resorted to an alternative (but equivalent [23]) representation of negative examples (cfr. Section 3.4), which improved the performance of our task. Table 1 summarizes the results obtained with the revisions by running 10 times each setting. The column *overlap* in Table 1 quantifies *how far* the repaired Petri net is from the original model (as a percentage of common facts). We can conclude that, on average, ILASP provides an optimal repair with an overlap exceeding 80% within a few minutes for medium-sized Petri nets. Moreover, in general, even removing a small amount of nodes from the search space produces a noticeable speed-up in finding solutions.

5 Conclusions

In this paper, we introduced an ILP-based approach to repair Petri net process models using ILASP. Our approach allows users to define edit operations (as logic programs), can be easily extended to handle noisy examples and to support different firing semantics. Experiments confirm this is a feasible approach for medium-sized models and models where domain knowledge is available. Our ASP-based approach makes it easy to seamlessly support many interesting features. For example, users can choose which parts of the input model should be kept as-is (e.g., unaffected by edit operations), as well as define constraints on the underlying bipartite graph topology of the Petri net (e.g., force the repaired model to be a WF-net, or having at most n outgoing arcs). Another interesting aspect, assuming a fixed set of edit operations, is that ILASP can process examples *incrementally* and asynchronously (making use of caches to store intermediate optimal solutions). This makes our approach a good basis for an interactive and (counter-)example-driven process model repair tool [7].

Acknowledgments. Francesco Chiariello was partially supported by the program France 2030 under the Grant agreement n°ANR-19-PI3A-0004. Antonio Ielo was partially supported by the Italian Ministry of Research (MUR) under PRIN projects PINPOINT - CUP H23C22000280006 and FAIR "Future AI Research" - Spoke 9 - WP9.1 - CUP H23C22000 860006.

References

[1] Wil M. P. van der Aalst. "Discovering Directly-Follows Complete Petri Nets from Event Data". In: *A Journey from Process Algebra via Timed Automata to Model Learning*. Vol. 13560. Lecture Notes in Computer Science. Springer, 2022, pp. 539–558.

[2] Wil M. P. van der Aalst. "The Application of Petri Nets to Workflow Management". In: *J. Circuits Syst. Comput.* 8.1 (1998), pp. 21–66.

[3] Wil M. P. van der Aalst and Josep Carmona, eds. *Process Mining Handbook*. Vol. 448. Lecture Notes in Business Information Processing. Springer, 2022.

[4] Wil M. P. van der Aalst and Christian Stahl. *Modeling Business Processes - A Petri Net-Oriented Approach*. Cooperative Information Systems series. MIT Press, 2011.

[5] Simone Agostinelli et al. "Process mining meets model learning: Discovering deterministic finite state automata from event logs for business process analysis". In: *Inf. Syst.* 114 (2023), p. 102180.

[6] Saadat Anwar, Chitta Baral, and Katsumi Inoue. "Encoding Petri Nets in Answer Set Programming for Simulation Based Reasoning". In: *Theory Pract. Log. Program.* 13 (2013).

[7] Abel Armas-Cervantes et al. "Incremental and Interactive Business Process Model Repair in Apromore". In: *BPM (Demos)*. Vol. 1920. CEUR Workshop Proceedings. CEUR-WS.org, 2017.

[8] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. "Answer set programming at a glance". In: *Commun. ACM* 54.12 (2011), pp. 92–103.

[9] Federico Chesani et al. "Exploiting Inductive Logic Programming Techniques for Declarative Process Mining". In: *Trans. Petri Nets Other Model. Concurr.* 2 (2009), pp. 278–295.

[10] Federico Chesani et al. "Optimising Business Process Discovery Using Answer Set Programming". In: *LPNMR*. Vol. 13416. Lecture Notes in Computer Science. Springer, 2022, pp. 498–504.

[11] Francesco Chiariello, Fabrizio Maria Maggi, and Fabio Patrizi. "ASP-Based Declarative Process Mining". In: *AAAI*. AAAI Press, 2022, pp. 5539–5547.

[12] Francesco Chiariello, Fabrizio Maria Maggi, and Fabio Patrizi. "From LTL on Process Traces to Finite-state Automata". In: *BPM (Demos / Resources Forum)*. Vol. 3469. CEUR Workshop Proceedings. CEUR-WS.org, 2023, pp. 127–131.

[13] Francesco Chiariello et al. “A Direct ASP Encoding for Declare”. In: *PADL*. Vol. 14512. Lecture Notes in Computer Science. Springer, 2024, pp. 116–133.

[14] Andrew Cropper et al. “Inductive logic programming at 30”. In: *Mach. Learn.* 111.1 (2022), pp. 147–172.

[15] Boudewijn van Dongen. *BPI Challenge 2020: Domestic Declarations*. 2020.

[16] Boudewijn F. van Dongen et al. “The ProM Framework: A New Era in Process Mining Tool Support”. In: *ICATPN*. Vol. 3536. Lecture Notes in Computer Science. Springer, 2005, pp. 444–454.

[17] Dirk Fahland and Wil M. P. van der Aalst. “Model repair - aligning process models to reality”. In: *Inf. Syst.* 47 (2015), pp. 220–243.

[18] Dirk Fahland and Wil M. P. van der Aalst. “Repairing Process Models to Reflect Reality”. In: *BPM*. Vol. 7481. Lecture Notes in Computer Science. Springer, 2012, pp. 229–245.

[19] Valeria Fionda and Gianluigi Greco. “LTL on Finite and Process Traces: Complexity Results and a Practical Reasoner”. In: *J. Artif. Intell. Res.* 63 (2018), pp. 557–623.

[20] Martin Gebser et al. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[21] Evelina Lamma et al. “Applying Inductive Logic Programming to Process Mining”. In: *ILP*. Vol. 4894. Lecture Notes in Computer Science. Springer, 2007, pp. 132–146.

[22] Mark Law. “Conflict-Driven Inductive Logic Programming”. In: *Theory Pract. Log. Program.* 23.2 (2023), pp. 387–414.

[23] Mark Law, Alessandra Russo, and Krysia Broda. “The complexity and generality of learning answer set programs”. In: *Artif. Intell.* 259 (2018), pp. 110–146.

[24] Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019.

[25] Fabrizio Maria Maggi et al. “Revising Process Models through Inductive Learning”. In: *Business Process Management Workshops*. Vol. 66. Lecture Notes in Business Information Processing. Springer, 2010, pp. 182–193.

[26] Artem Polyvyanyy et al. “Impact-Driven Process Model Repair”. In: *ACM Trans. Softw. Eng. Methodol.* 25.4 (2017), 28:1–28:60.

[27] Daniel Schuster, Sebastiaan J. van Zelst, and Wil M. P. van der Aalst. “Utilizing domain knowledge in data-driven process discovery: A literature review”. In: *Comput. Ind.* 137 (2022), p. 103612.

[28] Tijs Slaats, Søren Debois, and Christoffer Olling Back. “Weighing the Pros and Cons: Process Discovery with Negative Examples”. In: *BPM*. Vol. 12875. Lecture Notes in Computer Science. Springer, 2021, pp. 47–64.

[29] Mathias Weske. *Business Process Management - Concepts, Languages, Architectures, Third Edition*. Springer, 2019.