Incremental Declarative Process Mining

Massimiliano Cattafi, Evelina Lamma, Fabrizio Riguzzi, and Sergio Storari

ENDIF – Università di Ferrara – Via Saragat, 1 – 44100 Ferrara, Italy.
{massimiliano.cattafi,evelina.lamma,fabrizio.riguzzi,sergio.storari}@unife.it

Abstract. Business organizations achieve their mission by performing a number of processes. These span from simple sequences of actions to complex structured sets of activities with complex interrelation among them. The field of Business Processes Management studies how to describe, analyze, preserve and improve processes. In particular the subfield of Process Mining aims at inferring a model of the processes from logs (i.e. the collected records of performed activities). Moreover, processes can change over time to reflect mutated conditions, therefore it is often necessary to update the model. We call this activity Incremental Process Mining. To solve this problem, we modify the process mining system DPML to obtain IPM (Incremental Process Miner), which employs a subset of the $\mathcal{S}$CIFF language to represent models and adopts techniques developed in Inductive Logic Programming to perform theory revision. The experimental results show that is more convenient to revise a theory rather than learning a new one from scratch.

Keywords: Business Processes, Process Mining, Theory Revision

1 Introduction

In the current knowledge society, the set of business processes an organization performs in order to achieve its mission often represents one of the most important assets of the organization. It is thus necessary to be able to describe them in details, so that they can be stored and analyzed. In this way we can preserve and/or improve them. These problems are studied in the field of Business Processes Management (BPM) (see e.g. [1]).

Often organizations do not have a formal or precise description of the processes they perform. The knowledge necessary to execute the processes is owned by the individual workers but not by the organization as a whole, thus exposing it to possible malfunctions if a worker leaves.

However, modern information systems store all the actions performed by individual workers during the execution of a process. These action sequences are called traces and the set of all the traces recorded in a period of time is called a log. The Process Mining research area [2] proposes techniques for inferring a model of the process from a log.

Very often processes change over time to reflect mutated external or internal conditions. In this case, it is necessary to update their models. In particular,

given a process model and a new log, we want to modify the model so that it conforms also with the new log. We call this activity Incremental Process Mining. In this paper we show that revising an existing model may be more effective than learning a new model ex novo from the previous and new log. Moreover, in some cases the previous log may not be available, thus making model updating necessary.

We choose Logic Programming for the representation of traces and process models in order to exploit its expressiveness for the description of traces together with the wide variety of learning techniques available for it. An activity can be represented as a logical atom in which the predicate indicates the type of action and the arguments indicate the attributes of the action. One of the attributes is the time at which the action has been performed. Thus a trace can be represented as a set of instantiated atoms, i.e., a logical interpretation.

In order to represent process models, we use a subset of the $\mathcal{S}$CIFF language [3, 4]. A model in this language is a set of logical integrity constraints in the form of implications. Given a $\mathcal{S}$CIFF model and a trace, there exists an interpreter that checks whether the trace satisfies or not the model. Such a representation of traces and models is declarative in the sense that we do not explicitly state the allowed execution flows but we only impose high level constraints on them.

DPML is able to infer a $\mathcal{S}$CIFF theory from a set of positive and negative traces. Positive traces represent correct executions of the business process, while negative traces represent process executions that have been judged incorrect or undesirable.

Given that these traces are represented as logical interpretations and that $\mathcal{S}$CIFF integrity constraints are similar to logical clauses, DPML (Declarative Process Model Learner) [5] employs Inductive Logic Programming techniques [6] for learning from interpretations [7]. In particular, it modifies the ICL system [8] that learns sets of logical clauses from positive and negative interpretations.

In this paper, we present the system IPM (Incremental Process Miner) that faces the problem of revising an existing theory in the light of new evidence. This system is an adaptation of DPML and adopts techniques developed in Inductive Logic Programming (such as [9]) to perform theory revision.

IPM generalizes theories that do not satisfy new positive traces, as well as specializes theories that do not exclude new negative examples. To this purpose, we exploit the generalization operator presented in [5] for $\mathcal{S}$CIFF theories. Moreover, we define a specialization operator.

IPM is experimentally evaluated on processes regarding the management of a hotel and an electronic auction protocol. In the “hotel management” case the available traces are divided into two sets: one containing “old” traces and one containing “new” traces. Then two experiments are performed: in the first we learn a theory with DPML using the “old” traces and we revise the theory with IPM using the “new” traces, while in the latter we learn a theory with DPML from “old” and “new” traces. Then we compare the accuracy of the final theories obtained and the running time. A similar comparison is performed on the auction protocol, except that in this case the initial theory is not learned using some

"old" traces but it is a modified version of the actual model to simulate the revision of an imperfect theory written down by a user. Results associated to these experiments show that revising a theory is more efficient that inducing it from scratch. Moreover, the models obtained are more accurate on unseen data.

The paper is organized as follows. In Section 2 we recall the basic notions of Logic Programming, Inductive Logic Programming and Business Process Management. In Section 3 we discuss the representation of traces and models using Logic Programming. Section 4 illustrates the IPM algorithm. In Section 5 we report on the experiments performed. In Section 6 we discuss related works and we conclude with Section 7.

2 Preliminaries

We start by briefly recalling the basic concepts of Logic Programming, Inductive Logic Programming and Business Process Management.

2.1 Logic Programming

A *first order alphabet* Σ is a set of predicate symbols and function symbols (or functors) together with their arity. A *term* is either a variable or a functor applied to a tuple of terms of length equal to the arity of the functor. If the functor has arity 0 it is called a *constant*. An *atom* is a predicate symbol applied to a tuple of terms of length equal to the arity of the predicate. A *literal* is either an atom a or its negation $\neg a$. In the latter case it is called a *negative literal*. In logic programming, predicate and function symbols are indicated with alphanumeric strings starting with a lowercase character while variables are indicated with alphanumeric strings starting with an uppercase character.

A *clause* is a formula C of the form

$$h_1 \vee \ldots \vee h_n \leftarrow b_1, \ldots, b_m$$

where $h_1, \ldots, h_n$ are atoms and $b_1, \ldots, b_m$ are literals. A clause can be seen as a set of literals, e.g., C can be seen as

$$\{h_1, \ldots, h_n, \neg b_1, \ldots, \neg b_m\}.$$

In this representation, the disjunctions among the elements of the set are left implicit.

The form of a clause that is used in the following will be clear from the context. $h_1 \vee \ldots \vee h_n$ is called the *head* of the clause and $b_1, \ldots, b_m$ is called the *body*. We will use $head(C)$ to indicate either $h_1 \vee \ldots \vee h_n$ or $\{h_1, \ldots, h_n\}$, and $body(C)$ to indicate either $b_1, \ldots, b_m$ or $\{b_1, \ldots, b_m\}$, the exact meaning will be clear from the context. When $m = 0$, C is called a *fact*. When $n = 1$, C is called a *program clause*. When $n = 0$, C is called a *goal*. The conjunction of a set of literals is called a *query*. A clause is *range restricted* if all the variables that appear in the head appear as well in positive literals in the body.

A *theory* P is a set of clauses. A *normal logic program* P is a set of program clauses.

A term, atom, literal, goal, query or clause is *ground* if it does not contain variables. A *substitution* θ is an assignment of variables to terms: $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$. The *application of a substitution to a term, atom, literal, goal, query or clause* C, indicated with $C\theta$, is the replacement of the variables appearing in C and in θ with the terms specified in θ.

The *Herbrand universe* $H_U(P)$ is the set of all the terms that can be built with function symbols appearing in P. The *Herbrand base* $H_B(P)$ of a theory P is the set of all the ground atoms that can be built with predicate and function symbols appearing in P. A *grounding* of a clause C is obtained by replacing the variables of C with terms from $H_U(P)$. The grounding $g(P)$ of a theory P is the program obtained by replacing each clause with the set of all of its groundings. A *Herbrand interpretation* is a set of ground atoms, i.e. a subset of $H_B(P)$. In the following, we will omit the word Herbrand.

Let us now define the truth of a formula in an interpretation. Let I be an interpretation and ϕ a formula, ϕ is true in I, written $I \models \phi$ if

- $a \in I$, if ϕ is a ground atom a;
- $a \notin I$, if ϕ is a ground negative literal $\neg a$;
- $I \models a$ and $I \models b$, if ϕ is a conjunction $a \wedge b$;
- $I \models a$ or $I \models b$, if ϕ is a disjunction $a \vee b$;
- $I \models \psi\theta$ for all θ that assign a value to all the variables of $\mathbf{X}$ if $\phi = \forall \mathbf{X} \psi$;
- $I \models \psi\theta$ for a θ that assigns a value to all the variables of $\mathbf{X}$ if $\phi = \exists \mathbf{X} \psi$.

A clause C of the form

$$h_1 \vee \ldots \vee h_n \leftarrow b_1, \ldots, b_m$$

is a shorthand for the formula

$$\forall \mathbf{X} h_1 \vee \ldots \vee h_n \leftarrow b_1, \ldots, b_m$$

where $\mathbf{X}$ is a vector of all the variables appearing in C. Therefore, C is true in an interpretation I iff, for all the substitutions θ grounding C, if $I \models body(C)\theta$ then $I \models head(C)\theta$, i.e., if $(I \models body(C)\theta) \rightarrow (head(C)\theta \cap I \neq \emptyset)$. Otherwise, it is false. In particular, a program rule is true in an interpretation I iff, for all the substitutions θ grounding C, $(I \models body(C)\theta) \rightarrow h \in I$.

A theory P is true in an interpretation I iff all of its clauses are true in I and we write

$$I \models P.$$

If P is true in an interpretation I we say that I is a *model* of P. It is sufficient for a single clause of a theory P to be false in an interpretation I for P to be false in I

For normal logic programs, we are interested in deciding whether a query Q is a logical consequence of a theory P, expressed as

$$P \models Q.$$

This means that Q must be true in the model $M(P)$ of P that is assigned to P as its meaning by one of the semantics that have been proposed for normal logic programs (e.g. [10–12]).

For theories, we are interested in deciding whether a given theory or a given clause is true in an interpretation I. This can be achieved with the following procedure [13]. The truth of a range restricted clause C on a finite interpretation I can be tested by asking the goal ?-$body(C), \neg head(C)$ against a database containing the atoms of I as facts. By $\neg head(C)$ we mean $\neg h_1, \ldots, \neg h_m$. If the query fails, C is true in I, otherwise C is false in I.

In some cases, we are not given an interpretation I completely but we are given a set of atoms J and a normal program B as a compact way of indicating the interpretation $M(B \cup J)$. In this case, if B is composed only of range restricted rules, we can test the truth of a clause C on $M(B \cup J)$ by running the query ?-$body(C), \neg head(C)$ against a Prolog database containing the atoms of J as facts together with the rules of B. If the query fails C is true in $M(B \cup J)$, otherwise C is false in $M(B \cup J)$.

2.2 Inductive Logic Programming

Inductive Logic Programming (ILP) [6] is a research field at the intersection of Machine Learning and Logic Programming. It is concerned with the development of learning algorithms that adopt logic for representing input data and induced models. Recently, many techniques have been proposed in the field that were successfully applied to a variety of domains. Logic proved to be a powerful tool for representing the complexity that is typical of the real world. In particular, logic can represent in a compact way domains in which the entities of interest are composed of subparts connected by a network of relationships. Traditional Machine Learning is often not effective in these cases because it requires input data in the flat representation of a single table.

The problem that is faced by ILP can be expressed as follows:
Given:

- a space of possible theories $\mathcal{H}$;
- a set E^+ of positive example;
- a set E^- of negative examples;
- a background theory B.

Find a theory $H \in \mathcal{H}$ such that;

- all the positive examples are covered by H
- no negative example is covered by H

If a theory does not cover an example we say that it rules the example out so the last condition can be expressed by saying the "all the negative examples are ruled out by H".

The general form of the problem can be instantiated in different ways by choosing appropriate forms for the theories in input and output, for the examples and for the covering relation.

In the *learning from entailment* setting, the theories are normal logic programs, the examples are (most often) ground facts and the coverage relation is entailment, i.e., a theory H covers an example e iff

$$H \models e.$$

In the learning from interpretations setting, the theories are composed of clauses, the examples are interpretations and the coverage relation is truth in an interpretation, i.e., a theory H covers an example interpretation I iff

$$I \models H.$$

Similarly, we say that a clause C covers an example I iff $I \models C$.

In this paper, we concentrate on learning from interpretation so we report here the detailed definition:

Given:

- a space of possible theories $\mathcal{H}$;
- a set E^+ of positive interpretations;
- a set E^- of negative interpretations;
- a background normal logic program B.

Find a theory $H \in \mathcal{H}$ such that;

- for all $P \in E^+$, H is true in the interpretation $M(B \cup P)$;
- for all $N \in E^-$, H is false in the interpretation $M(B \cup N)$.

The background knowledge B is used to encode each interpretation parsimoniously, by storing separately the rules that are not specific to a single interpretation but are true for every interpretation.

The algorithm ICL [8] solves the above problem. It performs a covering loop (function ICL in Figure 1) in which negative interpretations are progressively ruled out and removed from the set E^-. At each iteration of the loop a new clause is added to the theory. Each clause rules out some negative interpretations. The loop ends when E^- is empty or when no clause is found.

The clause to be added in every iteration of the covering loop is returned by the procedure FindBestClause (Figure 1). It looks for a clause by using beam search with $p(\ominus|\overline{C})$ as a heuristic function, where $p(\ominus|\overline{C})$ is the probability that an example interpretation is classified as negative given that it is ruled out by the clause C. This heuristic is computed as the number of ruled out negative interpretations over the total number of ruled out interpretations (positive and negative). Thus we look for clauses that cover as many positive interpretations as possible and rule out as many negative interpretations as possible. The search starts from the clause $false \leftarrow true$ that rules out all the negative interpretations but also all the positive ones and gradually refines that clause.

The refinements of a clause are obtained by generalization. A clause C is *more general* than a clause D if the set of interpretations covered by C is a superset of those covered by D. This is true if $D \models C$. However, using logical implication

```
function ICL(E+, E−, B)
initialize H := ∅
do
    C := FindBestClause(E+, E−, B)
    if best clause C ≠ ∅ then
        add C to H
        remove from E− all interpretations that are false for C
while C ≠ ∅ and E− is not empty
return H

function FindBestClause(E+, E−, B)
initialize Beam := {false ← true}
initialize BestClause := ∅
while Beam is not empty do
    initialize NewBeam := ∅
    for each clause C in Beam do
        for each refinement Ref ∈ δ(C) do
            if Ref is better than BestClause then BestClause := Ref
            if Ref is not to be pruned then
                add Ref to NewBeam
                if size of NewBeam > MaxBeamSize then
                    remove worst clause from NewBeam
    Beam := NewBeam
return BestClause
```

Fig. 1. ICL learning algorithm

as a generality relation is impractical because of its high computational cost. Therefore, the syntactic relation of θ-*subsumption* is used in place of implication: D θ-*subsumes* C (written $D \geq C$) if there exist a substitution θ such that $D\theta \subseteq C$. If $D \geq C$ then $D \models C$ and thus C is more general than D. The opposite, however, is not true, so θ-subsumption is only an approximation of the generality relation. For example, let us consider the following clauses:

$C_1 = accept(X) \leftarrow true$
$C_2 = accept(X) \vee refusal(X) \leftarrow true$
$C_3 = accept(X) \leftarrow invitation(X)$
$C_4 = accept(alice) \leftarrow invitation(alice)$

Then $C_1 \geq C_2$, $C_1 \geq C_3$ but $C_2 \not\geq C_3$, $C_3 \not\geq C_2$ so C_2 and C_3 are more general than C_1, while C_2 and C_3 are not comparable. Moreover $C_1 \geq C_4$, $C_3 \geq C_4$ but $C_2 \not\geq C_4$ and $C_4 \not\geq C_2$ so C_4 is more general than C_1 and C_3 while C_2 and C_4 are not comparable.

From the definition of θ-subsumption, it is clear that a clause can be refined (i.e. generalized) by applying one of the following two operations on a clause

- adding a literal to the (head or body of the) clause
- applying a substitution to the clause

FindBestClause computes the refinements of a clause by applying one of the above two operations. Let us call $\delta(C)$ the set of refinements so computed for a clause C. The clauses are gradually generalized until a clause is found that covers all (or most of) the positive interpretations while still ruling out some negative interpretations.

The literals that can possibly be added to a clause are specified in the *language bias*, a collection of statements in an ad hoc language that prescribe which refinements have to be considered. Two languages are possible for ICL: $\mathcal{D}$LAB and rmode (see [14] for details). Given a language bias which prescribes that the body literals must be chosen among $\{invitation(X)\}$ and that the head disjuncts must be chosen among $\{accept(X), refusal(X)\}$, an example of refinements sequence performed by FindBestClause is the following:

$false \leftarrow true$
$accept(X) \leftarrow true$
$accept(X) \leftarrow invitation(X)$
$accept(X) \vee refusal(X) \leftarrow invitation(X)$

The refinements of clauses in the beam can also be pruned: a refinement is pruned if it is not statistical significant and if it cannot produce a value of the heuristic function larger than that of the best clause. As regards the first type of pruning, a statistical test is used, while as regards the second type, the best refinement that can be obtained is a clause that covers all the positive examples and rules out the same negative examples as the original clause.

When a new clause is returned by FindBestClause, it is added to the current theory. The negative interpretations that are ruled out by the clause are ruled out as well by the updated theory, so they can be removed from E^-.

2.3 Incremental Inductive Logic Programming

The learning framework presented in Section 2.2 assumes that all the examples are provided to the learner at the same time and that no previous model exists for the concepts to be learned. In some cases, however, the examples are not all known at the same time and an initial theory may be available. When a new example is obtained, one approach consists of adding the example to the previous training set and learning a new theory from scratch. This approach may turn out to be too inefficient, especially if the amount of previous examples is very high. An alternative approach consists in revising the existing theory to take into account the new example, in order to exploit as much as possible the computations already done. The latter approach is called Theory Revision and can be described by the following definition:
Given:

- a space of possible theories $\mathcal{H}$;
- a background theory B
- a set E^+ of previous positive example;
- a set E^- of previous negative examples;
- a theory H that is consistent with E^+ and E^-
- a new example e.

Find a theory $H' \in \mathcal{H}$ such that;

- H' is obtained by applying a number of transformations to H
- H' covers e if e is a positive example, or
- H' does not cover e if e is a negative example.

Theory Revision has been extensively studied in the learning from entailment setting of Inductive Logic Programming. In this case, examples are ground facts, the background theory is a normal logic program and the coverage relation is logical entailment. Among the systems that have been proposed for solving such a problem are: RUTH [15], FORTE [16] and Inthelex [9]

All these systems perform the following operations:

- given an uncovered positive example e, they generalize the theory T so that it covers it
- given a covered negative example e, they specialize the theory T so that it does not cover it

As an example of an ILP Theory Revision system, let us consider the algorithm of Inthelex that is shown in Figure 2. Function Generalize is used to revise the theory when the new example is positive. Each clause of the theory is considered in turn and is generalized. The resulting theory is tested to see whether it covers the positive example. Moreover, it is tested on all the previous negative examples to ensure that the clause is not generalized too much. As soon as a good refinement is found it is returned by the function.

Function Specialize is used to revise the theory when the new example is negative. Each clause of the theory involved in the derivation of the negative

```
function Generalize(E⁻, e, B, H)
repeat
    pick a clause C from H
    obtain a set of generalizations δ(C)
    for each clause C′ ∈ δ(C)
        let H′ := H \ {C} ∪ {C′}
        test H′ over e and over all the examples in E⁻
        if H′ cover e and does not cover any negative example then
            return H′
until all the clauses of H have been considered
// no generalization found
add a new clause to H that covers e and is consistent with E⁻
let H′ be the new theory
return H′


function Specialize(E⁺, e, B, H)
repeat
    pick a clause C used in the derivation of e in H
    obtain a set of specializations ρ(C)
    for each clause C′ ∈ δ(C)
        let H′ := H \ {C} ∪ {C′}
        test H′ over e and over all the examples in E⁻
        if H′ does not cover e and covers all positive examples then
            return H′
until all the clauses of H used in the derivation of e have been considered
// no specialization found
add e to H as an exception
let H′ be the new theory
return H′
```

Fig. 2. Inthelex Theory Revision algorithm

example is considered in turn and is specialized. The resulting theory is tested to see whether it rules out the negative example. Moreover, it is tested on all the previous positive examples to ensure that the clause is not specialized too much. As soon as a good refinement is found it is returned by the function.

While various systems exist for Theory Revision in the learning from entailment setting, to the best of our knowledge no algorithm has been proposed for Theory Revision in the learning from interpretation setting.

2.4 Business Process Management

The performances of an organization depend on how accurately and efficiently it enacts its business processes. Formal ways of representing business processes have been studied in the area of Business Processes Management (see e.g. [17]), so that the actual enactment of a process can be checked for compliance with a model.

Recently, the problem of automatically inferring such a model from data has been studied by many authors (see e.g. [18, 2, 19]). This problem has been called Process Mining or Workflow Mining. The data in this case consists of execution traces (or histories) of the business process. The collection of such data is made possible by the facility offered by many information systems of logging the activities performed by users.

Let us now describe in detail the problem that is solved by Process Mining. A *process trace* T is a sequence of events. Each event is described by a number of attributes. The only requirement is that one of the attributes describes the event type. Other attributes may be the executor of the event or event specific information.

An example of a trace is

$\langle a, b, c \rangle$

that means that activity a was performed first, then b and finally c.

A *process model* PM is a description of the process in a language that expresses the conditions a trace must satisfy in order to be compliant with the process, i.e., to be a correct enactment of the process. An interpreter of the language must exists that, when applied to a model PM and a trace T, returns yes if the trace is compliant with the description and false otherwise. In the first case we write $T \models PM$, in the second case $T \not\models PM$. A bag of process traces L is called a *log*. Often, in Process Mining, only compliant traces are used as input of the learning algorithm, see e.g. [18, 2, 19]. We consider instead the case in which we are given both compliant and non compliant traces. This is the case when we want distinguish successful process executions from unsuccessful ones.

The approaches presented in [18, 2, 19] aim at discovering complex and procedural process models, and differ by the structural patterns they are able to mine. While recognizing the extreme importance of such approaches, recently [20] pointed out the necessity of discovering declarative logic-based knowledge, in the form of process fragments or business rules/policies, from execution traces. Declarative languages seem to fit better complex, unpredictable processes, where a good balance between support and flexibility is of key importance.

[20] presents a graphical language for specifying process flows in a declarative manner. The language, called ConDec, leaves the control flow among activities partially unspecified by defining a set of constraints expressing policies/business rules for specifying either what is forbidden as well as mandatory in the process. Therefore, the approach is inherently open and flexible, because workers can perform actions if those are not explicitly forbidden. ConDec adopts an underlying semantics by means of Linear Temporal Logics (LTL), and can also be mapped onto the logic programming-based framework $\mathcal{S}$CIFF [3, 21] that provides a declarative language based on Computational Logic. In $\mathcal{S}$CIFF constraints are imposed on activities in terms of reactive rules (namely Integrity Constraints). Such reactive rules mention in their body occurring activities, i.e., *events*, and additional constraints on their variables. $\mathcal{S}$CIFF rules contain in their head expectations over the course of events. Such expectations can be positive, when a certain activity is required to happen, or negative, when a certain activity is forbidden to happen.

Most works in Process Mining deal with the discovery of procedural process models (such as Petri Nets or Event-driven Process Chains [22, 23]) from data. Recently, some works have started to appear on the discovery of logic--based declarative models: [24, 5, 25] study the possibility of inferring essential process constraints, easily understandable by business analysts and not affected by procedural details.

3 Representing Process Traces and Models with Logic

A process trace can be represented as a logical interpretation: each event is modeled with an atom whose predicate is the event type and whose arguments store the attributes of the action. Moreover, an extra argument is added to the atom indicating the position in the sequence. For example, the trace:

$\langle a, b, c\rangle$

can be represented with the interpretation

$\{a(1), b(2), c(3)\}$.

If the execution time is an attribute of the event, then the position in the sequence can be omitted.

Besides traces, we may have some general knowledge that is valid for all traces. We assume that this background information can be represented as a normal logic program B. The rules of B allow to complete the information present in a trace I: rather than simply I, we now consider $M(B \cup I)$, the model of the program $B \cup I$ according to one of the semantics for normal logic programs.

For example, consider the trace

$I = \{ask_price(bike, 1), tell_price(500, 2), buy(bike, 3)\}$

of a bike retail store and the background theory

$B = \{high_price(T) \leftarrow tell_price(P, T), P \geq 400\}$.

that expresses information regarding price perceptions by clients. Then $M(B \cup I)$ is

$\{ask_price(bike, 1), tell_price(500, 2), high_price(2), buy(bike, 3)\}$

in which the information available in the trace has been enlarged by using the background information.

The process language we consider was proposed in [5] and is a subset of the $\mathcal{S}$CIFF language, originally defined in [3, 4], for specifying and verifying interactions in open agent societies.

A process model in our language is a set of *integrity constraints* (ICs for short). An IC, C, is a logical formula of the form

$$Body \rightarrow \exists(ConjP_1) \vee \ldots \vee \exists(ConjP_n) \vee \forall\neg(ConjN_1) \vee \ldots \vee \forall\neg(ConjN_m) \quad (1)$$

where $Body$, $ConjP_i$ $i = 1, \ldots, n$ and $ConjN_j$ $j = 1, \ldots, m$ are conjunctions of literals built over event atoms, over predicates defined in the background or over built-in predicates such as $\leq, \geq, \ldots$. The variables appearing in the body are implicitly universally quantified with scope the entire formula. The quantifiers in the head apply to all the variables appearing in the conjunctions and not appearing in the body.

We will use $Body(C)$ to indicate $Body$ and $Head(C)$ to indicate the formula $\exists(ConjP_1) \vee \ldots \vee \exists(ConjP_n) \vee \forall\neg(ConjN_1) \vee \ldots \vee \forall\neg(ConjN_m)$ and call them respectively the *body* and the *head* of C. We will use $HeadSet(C)$ to indicate the set $\{ConjP_1, \ldots, ConjP_n, ConjN_1, \ldots, ConjN_m\}$.

$Body(C)$, $ConjP_i$ $i = 1, \ldots, n$ and $ConjN_j$ $j = 1, \ldots, m$ will be sometimes interpreted as sets of literals, the intended meaning will be clear from the context. We will call P *conjunction* each $ConjP_i$ for $i = 1, \ldots, n$ and N *conjunction* each $ConjN_j$ for $j = 1, \ldots, m$. We will call P *disjunct* each $\exists(ConjP_i)$ for $i = 1, \ldots, n$ and N *disjunct* each $\forall\neg(ConjN_j)$ for $j = 1, \ldots, m$.

An example of an IC is

$$\begin{aligned} & a(bob, T), T < 10 \\ \rightarrow & \exists(b(alice, T1), T < T1) \\ & \vee \\ & \forall\neg(c(mary, T1), T < T1, T1 < T + 10) \end{aligned} \quad (2)$$

The meaning of IC (2) is the following: if *bob* has executed action a at a time $T < 10$, then *alice* must execute action b at a time $T1$ later than T or *mary* must not execute action c for 9 time units after T. The disjunct $\exists(b(alice, T1), T < T1)$ stands for $\exists T1(b(alice, T1), T < T1)$ and the disjunct $\forall\neg(c(mary, T1), T < T1, T1 < T + 10)$ stands for $\forall T1\neg(c(mary, T1), T < T1, T1 < T + 10)$.

An *IC C is true in an interpretation* $M(B \cup I)$, written $M(B \cup I) \models C$, if, for every substitution θ for which $Body$ is true in $M(B \cup I)$, there exists a disjunct $\exists(ConjP_i)$ or $\forall\neg(ConjN_j)$ that is true in $M(B \cup I)$. If $M(B \cup I) \models C$ we say that the trace I is *compliant* with C. [5] showed that the truth of an IC in an interpretation $M(B \cup I)$ can be tested by running the query:

?-$Body, \neg(ConjP_1), \ldots \neg(ConjP_n), ConjN_1, \ldots, ConjN_m$

against a Prolog database containing the clauses of B and atoms of I as facts. If the N conjunctions in the head share some variables, then the following query must be issued

?-$Body, \neg(ConjP_1), \ldots \neg(ConjP_n), \neg(\neg(ConjN_1)), \ldots, \neg(\neg(ConjN_m))$
that ensures that the N conjunctions are tested separately without instantiating the variables.

Thus, for IC 2, the query is

?-$a(bob, T), T < 10, \neg(b(alice, T1), T < T1), \neg(\neg(c(mary, T1), T < T1, T1 < T + 10))$

If the query finitely fails, the IC is true in the interpretation. If the query succeeds, the IC is false in the interpretation. Otherwise nothing can be said. It is the user's responsibility to write the background B in such a way that no query generates an infinite loop. For example, if B is acyclic then a large class of queries will be terminating [26].

A *process model* H *is true in an interpretation* $M(B \cup I)$ if every IC of H is true in it and we write $M(B \cup I) \models H$. We also say that trace I is *compliant with* H.

The ICs we consider are more expressive than clauses, as can be seen from the query used to test them: for ICs, we have the negation of conjunctions, while for clauses we have only the negation of atoms. This added expressiveness is necessary for dealing with processes because it allows us to represent relations between the execution times of two or more activities.

4 Incremental Learning of ICs Theories

In order to induce a theory that describes a process, we must search the space of ICs. To this purpose, we need to define a generality order in such a space.

IC C is *more general than* IC D if C is true in a superset of the traces where D is true. If $D \models C$, then C is more general than D.

Similarly to the case of clauses, [5] defined the notion of θ-subsumption also for ICs.

Definition 1 (Subsumption). *An IC D θ-*subsumes *an IC C, written $D \geq C$, iff it exists a substitution θ for the variables in the body of D or in the N conjunctions of D such that*

- $Body(D)\theta \subseteq Body(C)$ *and*
- $\forall ConjP(D) \in HeadSet(D),\ \exists ConjP(C) \in HeadSet(C) : ConjP(C) \subseteq ConjP(D)\theta$ *and*
- $\forall ConjN(D) \in HeadSet(D),\ \exists ConjN(C) \in HeadSet(C) : ConjN(D)\theta \subseteq ConjN(C)$

For example, IC 2 is subsumed by the IC

$$\begin{aligned} & a(bob, 4) \\ \rightarrow & \exists(b(alice, T1), 4 < T1, T1 < 4 + 10) \\ & \vee \\ & \forall \neg(c(mary, 5), 4 < 5) \end{aligned} \tag{3}$$

with the substitution $\{T/4, T1/5\}$.

It was proved in [5] that implication and θ-subsumption for ICs share the same relation as in the case of clauses.

Theorem 1 ([5]). $D \geq C \Rightarrow D \models C$.

Thus, θ-subsumption can be used for defining a notion of generality among ICs, which can be used in learning algorithms.

In order to define a refinement operator, we must first define the language bias. We use a language bias that consists of a set of IC templates. Each template specifies

- a set of literals BS allowed in the body,
- a set of disjuncts HS allowed in the head. Each disjunct is represented as a couple $(Sign, DS)$ where
  - $Sign$ is either $+$ or $-$ and specifies where it is a P or an N disjunct,
  - DS is the set of literals allowed in the disjunct.

[5] defined a refinement operator from specific to general (upward operator) in the following way: given an IC D, the set of *upward refinements* $\delta(D)$ of D is obtained by performing one of the following operations

- adding a literal from BS to the body;
- removing a literal from a P disjunct in the head;
- adding a literal to an N disjunct in the head where the literal must be allowed by the language bias;
- adding a disjunct from HS to the head: the disjunct can be
  - a formula $\exists(d_1 \wedge \ldots \wedge d_k)$ where $DS = \{d_1, \ldots, d_k\}$ is the set of literals allowed by the IC template for D for a P disjunct,
  - a formula $\forall\neg(d)$ where d is allowed by the IC template for D for a N disjunct.

In order to perform theory revision, we also define a refinement operator from general to specific (downward operator). The operator inverts the operations performed in the upward operator, i.e., given an IC D, the set of *downward refinements* $\rho(D)$ of D is obtained by performing one of the following operations

- removing a literal from the body of D;
- adding a literal to a P disjunct in the head, the literal must be allowed by the language bias;
- removing a literal from an N disjunct in the head;
- removing a disjunct from the head when
  - it is a P disjunct $\exists(d_1 \wedge \ldots \wedge d_k)$ where $\{d_1, \ldots, d_k\}$ is the set of literals allowed by the IC template for D for the P disjunct,
  - it is an N disjunct containing a single literal $\forall\neg(d)$.

We define the algorithm for performing theory revision starting from the algorithm DPML (Declarative Process Model Learner) that is an adaptation of ICL [8]. DPML solves the following learning problem
Given

- a space of possible process models $\mathcal{H}$
- a set E^+ of positive traces;
- a set E^- of negative traces;
- a background normal logic program B.

Find: a process model $H \in \mathcal{H}$ such that

- for all $T^+ \in E^+$, $M(B \cup T^+) \models H$;
- for all $T^- \in E^-$, $M(B \cup T^-) \not\models H$;

If $M(B \cup T) \models C$ we say that IC C *covers* the trace T and if $M(B \cup T) \not\models C$ we say that C *rules out* the trace T.

DPML is obtained from ICL by using the testing procedure and the refinement operator defined for $\mathcal{S}$CIFF ICs in place of those for logical clauses.

The system IPM (Incremental Process Mines) modifies DPML in order to deal with theory revision. As in Section 2.3, we call $\mathcal{H}$ the space of possible theories, B the background theory, E^+ the set of previous positive examples, E^- the set of previous negative ones and T the theory (obtained by DPML or expressed by a human expert) we would like to refine to make it consistent with the new examples: $Enew^-$ and $Enew^+$. Figure 3 shows the IDPML algorithm.

The initial theory, together with old and new positive examples and old negative ones, is given as input to RevisePositive whose aim is to revise the theory in order to cover as many positive examples as possible. The output of RevisePositive is then given as input, together with all sets of examples, to ReviseNegative, whose revision tries to rule out the negative examples, and whose output is the overall revised theory.

RevisePositive cycles on new positive examples and finds out which constraints (if any) of the previous theory are violated for each example. An inner cycle generalizes all such constraints in order to make the theory cover the ruled out positive example. The generalization function performs a beam search with $p(\ominus|\overline{C})$ as the heuristic (see Section 2.2) and δ as the refinements operator (see Section 4). For theory revision, however, the beam is not initialized with the most specific constraint (i.e. $\{false \leftarrow true\}$) but with the violated constraint.

Since some of the previously ruled out negative examples may be again covered after the generalization process, ReviseNegative checks at first which negative examples, either old or new, are not ruled out. Then it selects randomly an IC from the theory and it performs a specialization cycle until no negative example is covered. The Specialize function is similar to the Generalize one with δ replaced with ρ as the refinement operator (see Section 4).

It is also possible that some negative examples can't be ruled out just by specializing existing constraints, so after ReviseNegative a covering loop (as the one of DPML) has to be performed on all positive examples and on the negative ones which are still to be ruled out.

5 Experiments

In this section we present some experiments that have been performed for investigating the effectiveness of IPM. In particular, we want to demonstrate that,

```
function IPM(T, E+, E-, Enew+, Enew-, B)
    H:= RevisePositive(T, E+, E-, Enew+, B)
    H:= ReviseNegative(H, E+, E-, Enew+, Enew-, B)
    H:=H∪ DPML(E+ ∪ Enew+, Covered(Enew-, H), B)
    return H

function RevisePositive(T, E+, E-, Enew+, B)
    foreach e+ ∈ Enew+
        VC:=FindViolatedConstraints(T, e+)
        T:= T − VC
        E+:=E+ ∪ {e+}
        foreach vc ∈ VC
            c:= Generalize(vc, E+, E-, B)
            T:= T ∪ {Best(c, vc)}
    return T

function Generalize(vc, E+, E-, B)
    Beam:={vc}
    BestClause:= ∅
    while Beam ≠ ∅
        foreach c ∈ Beam
            foreach ref of c
                BestClause:= Best(ref, c)
                Beam:= Beam ∪ {ref}
                if size(Beam) > MaxBeamSize
                    Beam:= Beam − {Worst(Beam)}
    return Beam

function ReviseNegative(T, E+, E-, Enew+, Enew-, B)
    Enew-:=TestNegative(T, E-, Enew-)
    E+:=E+ ∪ Enew+
    H:=∅
    while T ≠ ∅ ∧ Enew- ≠ ∅
        pick randomnly an IC c from T
        T:=T − {c}
        nc:= Specialize(c, E+, Enew-, B)
        H:=H ∪ {Best(c, nc)}
        Enew-:= Enew- − RuledOut(Enew-, Best(c, nc))
    return H
```

Fig. 3. IPM algorithm

given an initial theory H and a new set of examples $Enew$, it can be more beneficial to revise H in the light of $Enew$ than to learn a theory from $E \cup Enew$. Another use case consists in the revision of an (imperfect) theory written down by a user and its comparison with the theory learned from scratch using the same set of examples.

5.1 Hotel Management

Let's first consider a process model regarding the management of a hotel and inspired by [27]. We generated randomly a number of traces for this process, we classified them with the model and then we applied both DMPL and IDMPL.

The model describes a simple process of renting rooms and services in a hotel. Every process instance starts with the registration of the client name and her preferred way of payment (e.g., credit card). Data can also be altered at a later time (e.g the client may decide to use another credit card). During her stay, the client can require one or more room and laundry services. Each service, identified by a code, is followed by the respective registration of the service costs into the client bill. The cost of each service must be registered only if the service has been effectively provided to the client and it must be registered only once. The cost related to the nights spent in the hotel must be billed. It is possible for the total bill to be charged at several stages during the stay.

This process was modeled by using eight activities and eight constraints. Activities *register_client_data*, *check_out* and *charge* are about the check-in/check-out of the client and expense charging. Activities *room_service* and *laundry_service* log which services have been used by the client, while billings for each service are represented by separate activities. For each activity, a unique identifier is introduced to correctly charge the clients with the price for the services they effectively made use of.

Business related aspects of our example are represented as follows:

- (C.1) every process instance starts with activity *register_client_data*. No limits on the repetition of this activity are expressed, hence allowing alteration of data;
- (C.2) *bill_room_service* must be executed after each *room_service* activity, and *bill_room_service* can be executed only if the *room_service* activity has been executed before;
- (C.3) *bill_laundry_service* must be executed after each *laundry_service* activity, and *bill_laundry_service* can be executed only if the *laundry_service* activity has been executed before;
- (C.4) *check_out* must be performed in every process instance;
- (C.5) *charge* must be performed in every process instance;
- (C.6) *bill_nights* must be performed in every process instance.
- (C.7) *bill_room_service* must be executed only one time for each service identifier;
- (C.8) *bill_laundry_service* must be executed only one time for each service identifier;

The process model is composed by the following ICs:

$$
\begin{aligned}
(C.1)\ & true \\
& \rightarrow \exists(register_client_data(Trcd) \wedge Trcd = 1). \\
(C.2)\ & room_service(rs_id(IDrs), Trs) \\
& \rightarrow \exists(bill_room_service(rs_id(IDbrs), Tbrs) \wedge \\
& IDrs = IDbrs \wedge Tbrs > Trs). \\
& bill_room_service(rs_id(IDbrs), Tbrs) \\
& \rightarrow \exists(room_service(rs_id(IDrs), Trs) \wedge \\
& IDbrs = IDrs \wedge Trs < Tbrs). \\
(C.3)\ & laundry_service(la_id(IDls), Tls) \\
& \rightarrow \exists(bill_laundry_service(la_id(IDbls), Tbls) \wedge \\
& IDls = IDbls \wedge Tbls > Tls). \\
& bill_laundry_service(la_id(IDbls), Tbls) \\
& \rightarrow \exists(laundry_service(la_id(IDls), Tls) \wedge \\
& IDbls = IDls \wedge Tls < Tbls).
\end{aligned}
$$

$$
\begin{aligned}
(C.4)\ & true \\
& \rightarrow \exists(check_out(Tco)). \\
(C.5)\ & true \\
& \rightarrow \exists(charge(Tch)). \\
(C.6)\ & true \\
& \rightarrow \exists(bill_nights(Tbn)). \\
(C.7)\ & bill_room_service(rs_id(IDbrs1), Tbrs1) \\
& \rightarrow \forall\neg(bill_room_service(rs_id(IDbrs2), Tbrs2) \wedge \\
& IDbrs1 = IDbrs2 \wedge Tbrs2 > Tbrs1). \\
(C.8)\ & bill_laundry_service(la_id(IDbls1)), Tbls1) \\
& \rightarrow \forall\neg(bill_laundry_service(la_id(IDbls2), Tbls2) \wedge \\
& IDbls1 = IDbls2 \wedge Tbls2 > Tbls1).
\end{aligned}
$$

For this process, we randomly generated execution traces and we classified them with the above model. This was repeated until we obtained four training sets each composed of 2000 positive examples and 2000 negative examples. Each training set was randomly split into two subset, one containing 1500 positive and 1500 negative examples, and the other containing 500 positive and 500 negatives examples. The first subset is used for getting an initial theory, while the second is used for the revision process.

DPML was applied to each training sets with 3000 examples. The theories that were obtained were given as input to IPM together with the corresponding

dataset	revision				full dataset			
	time		accuracy		time		accuracy	
			μ	σ			μ	σ
1	4123		0.732539	0.0058	18963		0.702367	0.0068
2	4405		0.757939	0.0223	17348		0.686754	0.0269
3	6918		0.825067	0.0087	13480		0.662302	0.0180
4	3507		0.724764	0.0257	17786		0.679003	0.0248
	μ	σ	μ	σ	μ	σ	μ	σ
global	4738	1299	0.760	0.0433	16894	2057	0.682	0.0255

Table 1. Revision compared to learning from full dataset for the hotel scenario

1000 examples training set. Finally, DPML was applied to each of the complete 4000 examples training sets.

The models obtained by IPM and by DPML on a complete training set were then applied to each example of the other three training set. Accuracy is then computed as the number of compliant traces that are correctly classified as compliant plus the number of non-compliant traces that are correctly classified as non-compliant divided by the total number of traces.

In table 1 we show a comparison of time spent (in seconds) and resulting accuracies from the theory revision process and from learning based on the full dataset. The μ sub-columns for accuracy present means of results from tests on the three datasets not used for training, while in the σ one standard deviations can be found. The last row shows aggregated data for the correspondent columns.

As it can be noticed, in this case revising the theory to make it compliant with the new logs is faster than learning it again from scratch, and the accuracy of the results is higher.

5.2 Auction Protocol

Let us now consider an interaction protocol among agents participating in an electronic auction [28].

The auction is sealed bid: the auctioneer communicates the bidders the opening of the auction, the bidders answer with bids over the good and then the auctioneer communicates the bidders whether they have won or lost the auction.

The protocol is described by the following ICs [29].

$$\begin{aligned} & bid(B, A, Quote, TBid) \\ \rightarrow & \exists (openauction(A, B, TEnd, TDL, TOpen), \\ & TOpen < TBid, TBid < TEnd) \end{aligned} \tag{4}$$

This IC states that if a bidder sends the auctioneer a *bid*, then there must have been an *openauction* message sent before by the auctioneer and such that the

bid has arrived in time (before $TEnd$).

$$
\begin{aligned}
&openauction(A, B, TEnd, TDL, TOpen),\\
&bid(B, A, Quote, TBid),\\
&TOpen < TBid\\
\rightarrow&\exists(answer(A, B, lose, Quote, TLose),\\
&TLose < TDL, TEnd < TLose)\\
\vee&\exists(answer(A, B, win, Quote, TWin),\\
&TWin < TDL, TEnd < TWin)
\end{aligned} \tag{5}
$$

This IC states that if there is an *openauction* and a valid *bid*, then the auctioneer must answer with either *win* or *lose* after the end of the bidding time ($TEnd$) and before the deadline (TDL).

$$
\begin{aligned}
&answer(A, B, win, Quote, TWin)\\
\rightarrow&\forall\neg(answer(A, B, lose, Quote, TLose), TWin < TLose)
\end{aligned} \tag{6}
$$

$$
\begin{aligned}
&answer(A, B, lose, Quote, TLose)\\
\rightarrow&\forall\neg(answer(A, B, win, Quote, TWin), TLose < TWin)
\end{aligned} \tag{7}
$$

These two ICs state that the auctioneer can not answer both *win* and *lose* to the same bidder.

A graphical representation of the protocol is shown in Figure 4.

The traces have been generated in the following way: the first message is always *openauction*, the following messages are generated randomly between *bid* and *answer*. For *answer*, *win* and *lose* are selected randomly with equal probability. The bidders and auctioneer are always the same. The times are selected randomly from 2 to 10. Once a trace is generated, it is tested with the above ICs. If the trace satisfies all the ICs it is added to the set of positive traces, otherwise it is added to the set of negative traces. This process is repeated until 500 positive and 500 negative traces are generated for length 3, 4, 5 and 6. Five datasets are obtained in this way, containing each 2000 positive and 2000 negative traces.

We then considered 500 randomly selected traces (half positive and half negative). We applied both DPML and IPM to this dataset, the latter starting with a version of the model that was manually modified to simulate an imperfect theory written down by an user.

The results in Table 2 confirm those of Table 1: revision offers benefits both in time and accuracy.

6 Related Works

Process mining is an active research field. Notable works in such a field are [18, 22, 23, 19, 30, 31].

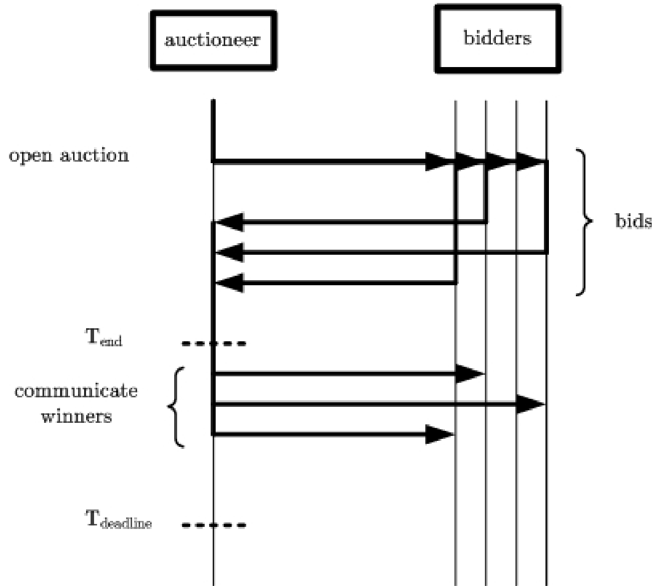

Fig. 4. Sealed bid auction protocol.

In particular in [30, 31] (partially) declarative specifications (thus closer to our work) are adopted.

In [31] activities in business process are seen as planning operators with pre-conditions and post-conditions. In order to explicitly express them, *fluents* besides activities (i.e., properties of the world that may change their truth value during the execution of the process) have to be specified. A plan for achieving the business goal is generated and presented to the user which has to specify whether each activity of the plan can be executed. In this way the system collects positive and negative examples of activities executions that are then used in a learning phase. Our work remains in the traditional domain of BPM in which the pre-conditions and post-conditions of activities are left implicit.

In [30] sets of process traces, represented by Petri nets, are described by high level process *runs*. Mining then performs a merge of runs regarding the same process and the outcome is a model which may contain sets of activities that must be executed, but for which no specific order is required. However, runs are are already very informative of the process model; in our work, instead, mining starts from traces, which are simply a sequence of events representing activity executions.

In [32] events used as negative examples are automatically generated in order to partially take away from the user the burden of having to classify activities. We are interested in the future to investigate automatic generation of negative traces.

	revision				full dataset			
dataset	time		accuracy		time		accuracy	
			μ	σ			μ	σ
1	820		0.962812	0.0043	1373		0.921687	0.0043
2	1222		0.962937	0.0043	1403		0.939625	0.0041
3	806		0.96375	0.0039	1368		0.923312	0.0044
4	698		0.961125	0.0018	1618		0.937375	0.0020
5	743		0.963875	0.0038	1369		0.92350	0.0042
	μ	σ	μ	σ	μ	σ	μ	σ
global	857	187	0.962	0.0039	1426	96	0.929	0.0086

Table 2. Revision compared to learning from dataset for the auction scenario

A useful survey about theory revision methods, including a brief description of common refinement operators and search strategies, can be found in [33]. With regard to the taxonomy proposed there, we deal with proper revision (not restructuring) both under the specializing and generalizing points of view.

In [34], the authors address structural process changes at run-time, once a process is implemented, in the context of adaptive process-aware information systems. Basically, adaptive systems are based on loosely specified models able to deal with uncertainty, i.e. able to be revised to cover unseen positive examples. The implemented process must be able to react to exceptions, i.e. it must be revised to rule-out unseen negative examples. Both kinds of revision must guarantee that compliant traces with the previous model are still compliant with the revised one. In [34], the authors consider process models expressed as Petri nets, where structural adaptation is based on high-level change patterns, previously defined in [35]. They review structural and behavioral correctness criteria needed to ensure the compliance of process instances to the changed schema. Then, they show how and under which criteria it is possible to support dynamic changes in the ADEPT2 system, also guaranteeing compliance. Similarly to them, we address the problem of updating a (declarative and rule-based, in our case) process model while preserving the compliance of process instances to the changed model. This is guaranteed, however, not by identifying correctness criteria, but rather by the theory revision algorithm itself. We think that their approach is promising, and subject for future work, in order to identify both change patterns to be considered (up to now we consider one generalization and one refinement operator only) and correctness criteria under which the revision algorithm might be improved.

7 Conclusions

In previous work we have presented the system DPML that is able to infer a process model composed of a set of logical integrity constraints starting from a log containing positive and negative traces.

In this paper we introduce the system IPM that modifies DMPL in order to be able to revise a theory in the light of new evidence. This allows to deal with the case in which the process changes over time and new traces are periodically collected. Moreover, it does not need to store all previous traces. IPM revises the current theory by generalizing it if it does not cover some positive traces and by specializing it if it covers some negative traces.

IPM has been tested on artificial data regarding a hotel management process and on an electronic auction protocol. The result shows that, when new evidence becomes available, revising the current theory in the light of the new evidence is faster than learning a theory from scratch, and the accuracy of the theories obtained in the first way is higher. This supports our claim that updating can be better that inducing a new theory.

In the future, we plan to perform more experiments in order to further analyze the performance difference between learning from scratch and revision. Moreover, we plan to investigate techniques for learning from positive only traces and for taking into account change patterns.

References

1. Dumas, M., Reichert, M., Shan, M., eds.: Business Process Management, 6th International Conference, BPM 2008. Volume 5240 of LNCS. Springer (2008)
2. van der Aalst, W.M.P., van Dongen, B.F., Herbst, J., Maruster, L., Schimm, G., Weijters, A.J.M.M.: Workflow mining: A survey of issues and approaches. Data Knowl. Eng. **47**(2) (2003) 237–267
3. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., P.Torroni: Verifiable agent interaction in abductive logic programming: The sciff framework. ACM Trans. Comput. Log. **9**(4) (2008)
4. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: An abductive interpretation for open societies. In Cappelli, A., Turini, F., eds.: 8th Congress of the Italian Association for Artificial Intelligence (AI*IA 2003). Volume 2829 of LNAI., Springer Verlag (2003)
5. Lamma, E., Mello, P., Riguzzi, F., Storari, S.: Applying inductive logic programming to process mining. In: Inductive Logic Programming, 17th International Conference. Number 4894 in Lecture Notes in Artificial Intelligence, Heidelberg, Germany, Springer (2008) 132–146
6. Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. Journal of Logic Programming **19/20** (1994) 629–679
7. Raedt, L.D., Dzeroski, S.: First-order jk-clausal theories are pac-learnable. Artif. Intell. **70**(1-2) (1994) 375–392
8. De Raedt, L., Van Laer, W.: Inductive constraint logic. In: Algorithmic Learning Theory, 6th Conference. Volume 997 of LNAI., Springer Verlag (1995)
9. Esposito, F., Semeraro, G., Fanizzi, N., Ferilli, S.: Multistrategy theory revision: Induction and abduction in inthelex. Machine Learning **38**(1-2) (2000) 133–156
10. Clark, K.L.: Negation as failure. In: Logic and Databases. Plenum Press (1978)
11. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the Fifth International Conference and Symposium on Logic Programming. (1988) 1070–1080

12. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM **38**(3) (1991) 620–650
13. De Raedt, L., Dehaspe, L.: Clausal discovery. Machine Learning **26**(2-3) (1997) 99–146
14. Van Laer, W.: ICL manual Available at: `http://www.cs.kuleuven.be/~ml/ACE/DocACEuser.pdf`.
15. Adé, H., Malfait, B., Raedt, L.D.: Ruth: an ilp theory revision system. In Ras, Z.W., Zemankova, M., eds.: Methodologies for Intelligent Systems, 8th International Symposium, ISMIS '94, Charlotte, North Carolina, USA, October 16-19, 1994, Proceedings. Volume 869 of Lecture Notes in Computer Science., Springer (1994) 336–345
16. Richards, B.L., Mooney, R.J.: Automated refinement of first-order horn-clause domain theories. Machine Learning **19**(2) (1995) 95–131
17. Georgakopoulos, D., Hornick, M.F., Sheth, A.P.: An overview of workflow management: From process modeling to workflow automation infrastructure. Distributed and Parallel Databases **3**(2) (1995) 119–153
18. Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. In: Proceedings of the 6th International Conference on Extending Database Technology, EDBT'98. Volume 1377 of LNCS., Springer (1998) 469–483
19. Greco, G., Guzzo, A., Pontieri, L., Saccà, D.: Discovering expressive process models by clustering log traces. IEEE Trans. Knowl. Data Eng. **18**(8) (2006) 1010–1027
20. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In: 2006 International Business Process Management Workshops. Volume 4103 of LNCS., Springer (2006) 169–180
21. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative specification and verification of service choreographies. ACM Transactions on The Web (2009)
22. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Trans. Knowl. Data Eng. **16**(9) (2004) 1128–1142
23. van Dongen, B.F., van der Aalst, W.M.P.: Multi-phase process mining: Building instance graphs. In: 23rd International Conference on Conceptual Modeling. Volume 3288 of LNCS., Springer (2004) 362–376
24. Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Inducing declarative logic-based models from labeled traces. In: Proceedings of the 5th International Conference on Business Process Management. Number 4714 in Lecture Notes in Computer Science, Heidelberg, Germany, Springer (2007) 344–359
25. Chesani, F., Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Exploiting inductive logic programming techniques for declarative process mining. LNCS Transactions on Petri Nets and Other Models of Concurrency, ToPNoC II **5460** (2009) 278–295
26. Apt, K.R., Bezem, M.: Acyclic programs. New Generation Comput. **9**(3/4) (1991) 335–364
27. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: Declare: Full support for loosely-structured processes. In: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), IEEE Computer Society (2007) 287–300
28. Chavez, A., Maes, P.: Kasbah: An agent marketplace for buying and selling goods. In: Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM-96), London (April 1996) 75–90

29. Chesani, F.: Socs protocol repository Available at: `http://edu59.deis.unibo.it:8079/SOCSProtocolsRepository/jsp/index.jsp`.
30. Desel, J., Erwin, T.: Hybrid specifications: looking at workflows from a run-time perspective. Int. J. Computer System Science & Engineering **15**(5) (2000) 291 – 302
31. Ferreira, H.M., Ferreira, D.R.: An integrated life cycle for workflow management based on learning and planning. Int. J. Cooperative Inf. Syst. **15**(4) (2006) 485–505
32. Goedertier, S.: Declarative techniques for modeling and mining business processes. PhD thesis, Katholieke Universiteit Leuven, Faculteit Economie en Bedrijfswetenschappen (2008)
33. Wrobel, S.: First order theory refinement. In Raedt, L.D., ed.: Advances in Inductive Logic Programming. IOS Press, Amsterdam (1996) 14 – 33
34. Reichert, M., Rinderle-Ma, S., Dadam, P.: Flexibility in process-aware information systems. T. Petri Nets and Other Models of Concurrency **2** (2009) 115–135
35. Mutschler, B., Reichert, M., Rinderle, S.: Analyzing the dynamic cost factors of process-aware information systems: A model-based approach. In: CAiSE. (2007) 589–603