Applying Inductive Logic Programming to Process Mining

Evelina Lamma, Paola Mello, Fabrizio Riguzzi, and Sergio Storari

Dip. di Ingegneria – Università di Ferrara – Via Saragat, 1 – 44100 Ferrara, Italy.
{evelina.lamma,fabrizio.riguzzi,sergio.storari}@unife.it
DEIS – Università di Bologna – Viale Risorgimento, 2 – 40136 Bologna, Italy.
pmello@deis.unibo.it

Abstract. The management of business processes has recently received a lot of attention. One of the most interesting problems is the description of a process model in a language that allows the checking of the compliance of a process execution (or trace) to the model. In this paper we propose a language for the representation of process models that is inspired to the SCIFF language and is an extension of clausal logic. A process model is represented in the language as a set of integrity constraints that allow conjunctive formulas as disjuncts in the head. We present an approach for inducing these models from data: we define a subsumption relation for the integrity constraints, we define a refinement operator and we adapt the algorithm ICL to the problem of learning such formulas. The system has been applied to the problem of inducing the model of a sealed bid auction and of the NetBill protocol. The data used for learning and testing were randomly generated from a correct model of the process.

Keywords: Process Mining, Learning from Interpretations, Business Processes, Interaction Protocols

1 Introduction

Every organization performs a number of *business processes* in order to achieve its mission. Complex organizations are characterized by complex processes, involving many people, activities and resources. 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. [1]), 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. [2–4]). 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.

In this paper, we propose a novel representation language for describing process models and an approach to Process Mining that uses learning from interpretations techniques from ILP. The language is inspired to the $\mathcal{S}$CIFF one [5] and extends clausal logic by allowing more complex formulas as disjuncts in the head of clauses.

We show how an execution trace can be represented as an interpretation and how we can use our language to check its compliance with the model. Thus we can cast a process mining problem as a learning from interpretation problem. In particular, we considered the discriminant problem that is solved by ICL [6], where we have positive and negative interpretations and we want to find a clausal theory that discriminates the two. In our case we assume that we have compliant and non compliant traces of execution of a process and we want to find a theory that accurately classifies a new trace as compliant or non compliant.

We differ from traditional process mining research in three respects: first we perform mining from both compliant and non compliant traces, while traditionally only compliant traces are considered; second we learn a declarative representation of a process model, while usually more procedural representation have been induced, such as Petri nets, and third, we are able to consider structured atomic activities, thanks to the first order representation.

The fact of having positive and negative traces is not commonly considered in the literature on process mining but it is interesting in a variety of cases: for example, a bank may divide its transactions into fraudulent and normal ones and may desire to learn a model that is able to discriminate the two. In general, an organization may have two or more sets of process executions and may want to understand in what sense they differ.

The use of a declarative language for process models is advocated by other authors as well [7]. The use of a structured representation allows the system to take into account many different properties of activities that would otherwise be overlooked.

The paper is organized as follows. Section 2 discusses how we represent execution traces with logic programming and describes the language used to represent process models. Section 3 presents the learning technique we have adopted for performing Process Mining. Section 4 reports on the experiments performed. Section 5 discusses related works and finally Section 6 concludes the paper.

2 A Representation for Process Traces and Models

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

a, b, d

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

A *process model* PM is a formula in a language. An interpreter of the language must exists that, when applied to a model PM and a trace t, returns answer 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*. Usually, in Process Mining, only compliant traces are used as input of the learning algorithm, see e.g. [2–4]. We consider instead the case where we are given both compliant and non compliant traces.

A process trace can be represented as an 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:

a, b, d

can be represented with the interpretation

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

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

Besides the trace, we may have some general knowledge that is valid for all traces. This information will be called background knowledge and we assume that it can be represented as a normal logic program B. The rules of B allow to complete the information present in a trace t: rather than simply t, we now consider $M(B \cup t)$, the model of the program $B \cup t$ according to Clark's completion [8].

The process language we consider is a subset of the $\mathcal{S}$CIFF language, originally defined in [9, 5], for specifying and verifying interaction in open agent societies.

A process model in our language is a set of Integrity Constraints (ICs). 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. The quantifiers in the head apply to all the variables not appearing in the body. The variables of the body are implicitly universally quantified with scope the entire formula.

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

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

The meaning of the 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.

An IC C is true in an interpretation $M(B \cup t)$, written $M(B \cup t) \models C$, if, for every substitution θ for which *Body* is true in $M(B \cup t)$, there exists a disjunct $\exists(ConjP_i)$ or $\forall\neg(ConjN_j)$ that is true in $M(B \cup t)$. If $M(B \cup t) \models C$ we say that the trace t is *compliant* with C.

Similarly to what has been observed in [10] for disjunctive clauses, the truth of an IC in an interpretation $M(B \cup t)$ can be tested by running the query:

$? - Body, not(ConjP_1), \ldots not(ConjP_n), ConjN_1, \ldots, ConjN_m$

in a database containing the clauses of B and atoms of t as facts.

If the N conjunctions in the head share some variables, then the following query must be issued

$? - Body, not(ConjP_1), \ldots not(ConjP_n),$

$not(not(ConjN_1)), \ldots, not(not(ConjN_m))$

that ensures that the N conjunctions are tested separately without instantiating the variables.

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 the queries will be terminating for a large class of queries [11].

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

The ICs we consider are more expressive than logical 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 expressivity is necessary for dealing with processes because it allows us to represent relations between the execution times of two or more activities.

3 Learning ICs Theories

In order to learn 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.

Definition 1 (Subsumption). *An IC D 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*
- $\forall ConjN(D) \in HeadSet(D)$, $\exists ConjN(C) \in HeadSet(C) : ConjN(D)\theta \subseteq ConjN(C)$

Theorem 1. $D \geq C \Rightarrow D \models C$.

Proof. We must prove that all the models of D are also models of C. Let θ be the substitution with which D subsumes C. Consider a model i of D. If $\nexists\delta$ such that $Body(C)\delta$ is true in i, then C is true in i.

If $\exists\delta$ such that $Body(C)\delta$ is true in i, then $Body(D)\theta\delta$ will be true in i because $Body(D)\theta\delta \subseteq Body(C)\delta$. So there must be a disjunct of $Head(D)\theta\delta$ that is true in i.

Suppose that the disjunct $\exists(ConjP(D))$ of D is such that $\exists(ConjP(D)\theta\delta)$ is true in i: C will contain a disjunct $\exists ConjP(C)$ such that $ConjP(C) \subseteq ConjP(D)\theta$, thus $ConjP(C)\delta \subseteq ConjP(D)\theta\delta$ and it holds that $\exists(ConjP(C)\delta)$.

Suppose that the disjunct $\forall\neg(ConjN(D))$ of D is such that $\forall\neg(ConjN(D)\theta\delta)$ is true in i: C will contain a disjunct $\forall\neg(ConjN(C))$ such that $ConjN(D)\theta \subseteq ConjP(C)$, thus $ConjN(D)\delta\theta \subseteq ConjP(C)\delta$ and it holds that $\forall\neg(ConjP(C)\delta)$.

Thus i is also a model of C

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. For each disjunct, the template specifies:
  - whether it is a P or an N disjunct,
  - the set of literals allowed in the disjunct.

Thus we can define a refinement operator in the following way: given an IC D, the set of refinements $\rho(D)$ of D is obtained by performing one of the following operations

- adding a literal from the IC template for D to the body;
- adding a disjunct from the IC template for D to the head: the disjunct can be
  - a conjunction $d_1 \wedge \ldots \wedge d_k$ where $\{d_1, \ldots, d_k\}$ is the set of literals allowed by the IC template for D for the P disjunct,
  - a literal d that is allowed by the IC template for D for a N disjunct;
- removing a literal from a P disjunct in the head;
- adding a literal to an N disjunct in the head. The literal must be allowed by the language bias.

The learning problem we consider is an adaptation to ICs of the learning from interpretation setting of ILP:
Given

- a space of possible process models $\mathcal{H}$
- a set I^+ of positive interpretations;
- a set I^- of negative interpretations;
- a definite clause background theory B.

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

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

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

In order to solve the problem, we propose the algorithm DPML (Declarative Process Model Learner) that is an adaptation of ICL [6].

```
function DPML($I^+, I^-, B$)
initialize $H := \emptyset$
do
    $C :=$ FindBestIC($I^+, I^-, B$)
    if $C \neq \emptyset$ then
        add $C$ to $H$
        remove from $I^-$ all interpretations that are false for $C$
while $C \neq \emptyset$ and $I^-$ is not empty
return $H$

function FindBestIC($I^+, I^-, B$)
initialize $Beam := \{false \leftarrow true\}$
initialize $BestIC := \emptyset$
while $Beam$ is not empty do
    initialize $NewBeam := \emptyset$
    for each IC $C$ in $Beam$ do
        for each refinement $Ref$ of $C$ do
            if $Ref$ is better than $BestIC$ then $BestIC := Ref$
            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 $BestIC$
```

Fig. 1. DPML learning algorithm

DPML performs a covering loop (function DPML, Figure 1) in which negative interpretations are progressively ruled out and removed from the set I^-. At each iteration of the loop a new IC is added to the theory. Each IC rules out some negative interpretations. The loop ends when I^- is empty or when no IC is found.

The IC to be added in every iteration of the covering loop is returned by the procedure FindBestIC (Figure 1). It looks for an IC by using beam search with $p(\ominus|\overline{C})$ as a heuristic function, where $p(\ominus|\overline{C})$ is the probability that an input trace is negative given that is ruled out by the IC C. This heuristic is computed as the number of ruled out negative traces over the total number of ruled out traces (positive and negative). Thus we look for formulas that cover as many positive traces as possible and rule out as many negative traces as possible. The search starts from the IC $false \leftarrow true$ that rules out all the negative traces but also all the positive traces and gradually refines that clause in order to make it more general. Even if the heuristic value of $false \leftarrow true$ is $p(\ominus)$, i.e. the fraction of negative traces in the training set, this IC is initially assigned an heuristic of 0 so that it is not considered better of any other IC. $MaxBeamSize$ is a user-defined constant storing the maximum size of the beam.

The heuristic of each generated refinement is compared with the one of the best IC found so far and, if the value is higher, the best IC is updated. At the end of the refinement cycle, the best IC found so far is returned.

DPML differs from ICL in three respects: we use a different testing procedure, a different refinement operator and a simpler pruning. As regards the refinement operator, $\mathcal{D}$LAB does not allow the possibility of having a conjunction inside negation and it does not allow the deletion of literals from a refinement.

As regards pruning, we do not prune the IC that are not statistically significant but we prune only the refinements that can not become better than the current best clause. We decided to do so because we observed that statistical significance has a low impact on experiments.

4 Experiments

We consider two interaction protocols among agents: an electronic auction protocol [12] and the NetBill protocol [13]. In both cases, we start from a set of ICs describing the protocol, and we randomly generat some traces for the protocol. They are then classified according to the model and are used for learning. For testing, we use a separate set of randomly generated traces.

The first protocol we consider is a sealed bid auction where 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 [14].

$$\begin{aligned} & bid(B, A, Quote, TBid) \\ \rightarrow & \exists(openauction(A, B, TEnd, TDL, TOpen), \\ & TOpen < TBid, TBid < TEnd) \end{aligned} \tag{3}$$

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 $TEnd$).

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

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).

$$\begin{aligned}
&answer(A, B, win, Quote, TWin)\\
\rightarrow&\forall\neg(answer(A, B, lose, Quote, TLose), TWin < TLose)
\end{aligned} \tag{5}$$

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

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 2.

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. Thus overall there are 2000 positive traces and 2000 negative traces.

NetBill is a security and transaction protocol optimized for the selling and delivery of low-priced information goods, such as software or journal articles, across the Internet. The protocols involves three parties: the customer, the merchant and the NetBill server. Here is an outline of the NetBill protocol (see Figure 3) :

1. the customer requests a price for a good from the merchant;
2. the merchant answers with a price for the good;
3. the customer can accept the offer, refuse it or make another request to the merchant, thus initiating a new negotiation and going back to step 2;
4. if the customer accepts the offer, it tells it to the merchant;
5. the merchant delivers the good to the customer encrypted with key K;

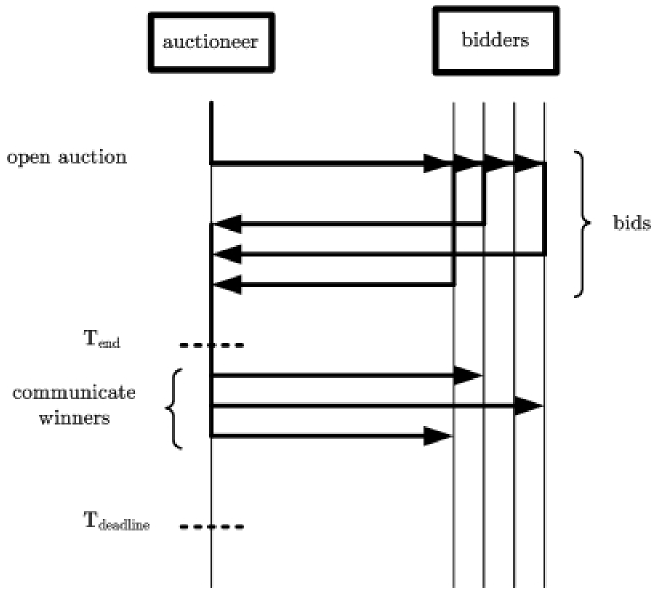

Fig. 2. Sealed bid auction protocol.

6. the customer prepares an electronic purchase order (EPO) digitally signed by her and sends it to the merchant;
7. the merchant countersigns the EPO and sends it and the value of K to the NetBill server;
8. the NetBill server checks the signature and counter-signature on the EPO. If customer's account contains enough funds, the NetBill server transfers the price from the customer's account to the merchant's account. The NetBill server then prepares a signed receipt that includes the value K, and it sends this receipt to the merchant;
9. the merchant records the receipt and forwards it to the customer (who can then decrypt her encrypted goods).

The NetBill protocol is represented using 19 ICs [14]. One of them is

$$\begin{aligned}
&request(C, M, good(G, Q), Nneg, Trq),\\
&present(M, C, good(G, Q), Nneg, Tp), Trq \leq Tp\\
\rightarrow&\exists(accept(C, M, good(G, Q), Ta), Tp \leq Ta)\\
\vee&\exists(refuse(C, M, good(G, Q), Trf), Tp \leq Trf)\\
\vee&\exists(request(C, M, good(G, Qrql), Nnegl, Trql), Tp \leq Trql)
\end{aligned} \tag{7}$$

This IC states that if there has been a *request* from the customer to the merchant and the merchant has answered with the same price, then the customer should either *accept* the offer, *refuse* the offer or start a new negotiation with a *request.*

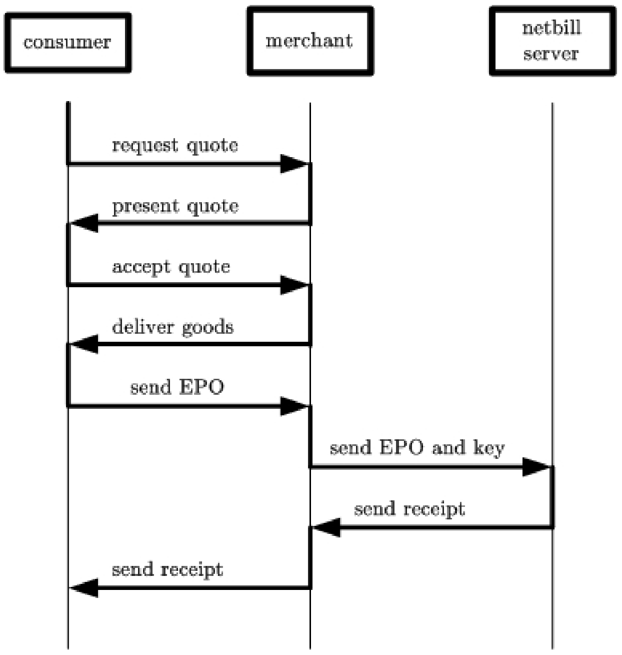

Fig. 3. NetBill transaction protocol.

The traces have been generated randomly in two stages: first the negotiation phase is generated and then the transaction phase. In the negotiation phase, we add to the end of the trace a *request* or *present* message with its arguments randomly generated with two possible values for Q (quote). The length of the negotiation phase is selected randomly between 2 and 5. After the completion of the negotiation phase, either an *accept* or a *refuse* message is added to the trace and the transaction phase is entered with probability 4/5, otherwise the trace is closed.

In the transaction phase, the messages *deliver*, *epo*, *epo_and_key*, *receipt* and *receipt_client* are added to the trace. With probability 1/4 a message from the whole trace is then removed.

Once a trace has been generated, it is classified with the ICs of the correct model and assigned to the set of positive or negative traces depending on the result of the test. The process is repeated until 2000 positive traces and 2000 negative traces have been generated.

Five training sets have been generated for the auction protocol and five for the NetBill protocol. Then DPML and the α-algorithm [15] have been applied to each of them. The α-algorithm is one of the first process mining algorithms and it induces Petri nets. We used the implementation of it available in the ProM suite [16]. Since the α-algorithm takes as input a single set of traces, we have provided it with the positive traces only.

The language bias that was used for the auction protocol is the following:

- BS contains two sets of instances of each action (open auction, bid, answer win and answer lose), with the instances of each set having the same variables,
- HS contains a P conjunction for open auction, answer win and answer lose and an N conjunction for open auction, answer win and answer lose. The conjunctions for open auction will contain atoms for the predicate *less* that compare each of its time arguments with the times of the literals in the body. The conjunctions for answer will contain atoms for the predicate *less* that compare its time to the time arguments of the literals in the body

For example, BS will contain
openauction(f,taxi1,TEnd,TDead,T)
and
openauction(f,taxi1,TEnd1,TDead1,T1)
and HS will contain the P conjunction:
{*answer(f,taxi1,lose,taxi2station,Price3,T2), lessp(T,T2), lessp(T1,T2), lessp(TEnd,T2), lessp(TDead,T2), lessp(TEnd1,T2), lessp(TDead1,T2), lessp(T2,TDead), lessp(T2,TDead1)*}
where *lessp*(A, B) is a predicate that fails if one of its two arguments is not instantiated and is equal to $A < B$ otherwise. In this way, if one of its arguments is not instantiated, the disjunct can not be true and the learning algorithm must either add the literal with the variable to the body or remove the *lessp* atom.

The language bias that was used for NetBill is the following:

- BS contains two sets of instances of each action (request, present, accept, deliver, send epo, send epo and key, receipt and receipt to client), with the instances of each set having the same variables,
- HS contains a P conjunction and an N conjunction for each action. The conjunctions will contain atoms for the predicate *less* that compare the time argument of the action with the times of the actions in the body plus atoms for the predicate *equal* for comparing the quote of the action in the head with those in the body

For example, BS will contain
request(c,m,good(software,Q),T)
and
request(c,m,good(software,Q1),T1)
and HS will contain the P conjunction:
{*hap(tell(c,m,request(software,Q2,n1)),T2), lessp(T,T2), lessp(T1,T2), lessp(T2,T), lessp(T2,T1), equalp(Q2,Q), equalp(Q2,Q1)*}
where *lessp*(A, B) is defined as before and *equalp*(A, B) is false if one of the arguments is not instantiated and is equal to $A =:= B$ otherwise.

The learned theories have been tested on five testing set generated with the same procedure used for the training set but with different seeds for the random functions. For the α-algorithm, the Petri net learned from positive traces only was used to replay the positive and negative test traces. The accuracy is given by the number of positive traces that are replayed correctly plus the number of negative traces not replayed correctly divided by the total number of test traces.

The average accuracy and the standard deviation of DPML and of the α-algorithm are shown in Table 1. The table shows also the the average number of ICs learned by DPML.

Table 1. Results of the experiments.

	DPML			α-algorithm	
Experiment	Av. acc.	St. dev.	Av. # ICs	Av. acc.	St. dev.
Auction	97.00%	3.7%	4	-	-
NetBill	94.65%	2.5%	9	66.81%	0.24%

The average time taken by DPML are 0.435 hours for Auction and 1.875 hours for NetBill on a Pentium M 2.00 GHz machine. The average time taken by the α-algorithm is under one minute for both datasets.

The α-algorithm was not applied to the auction protocol since it has no way of testing the satisfaction of the deadline, given that it considers an atomic model of the activities.

As can be seen, DPML outperforms the α-algorithm on NetBill in terms of accuracy, even if at the expense of a high computational cost. In order to find out how scalable is our approach, we run a series of experiments with increasing number of traces, from 500 up to 2000. The execution times on a machine with a Core Duo 1.86 GHz are shown in Figure 4. The graph shows that the execution

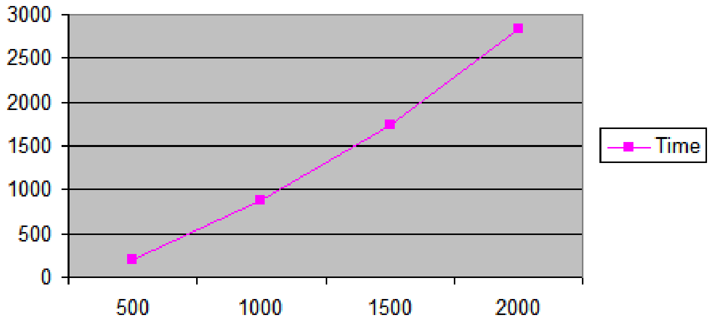

Fig. 4. Scalability of the DPML.

time increases nearly linearly with the number of traces.

5 Related Works

The integrity constraints presented in this paper are inspired to the integrity constraints of the $\mathcal{S}$CIFF language [9, 5]. For example, IC (2) would be written in the $\mathcal{S}$CIFF language as

$$\begin{aligned} & \mathbf{H}(a(bob), T) \wedge T < 10 \\ \rightarrow & \mathbf{E}(b(alice), T1) \wedge T < T1 \\ & \vee \\ & \mathbf{EN}(c(mary), T1) \wedge T < T1 \wedge T1 < T + 10 \end{aligned} \tag{8}$$

where **H** stands for "happened", **E** for "expected to happen" and **EN** for "expected not to happen".

The $\mathcal{S}$CIFF language allows for much more complex ICs than the ones considered in this paper and is equipped with an abductive proof procedure for testing the compliance of a trace. In particular, the $\mathcal{S}$CIFF language allows for the combination of variables with different quantification in the same head disjunct. We focused on a subset for its nice computational properties.

[2] introduced the idea of applying process mining to workflow management. The authors propose an approach for inducing a process representation in the form of a directed graph encoding the precedence relationships.

[15] presents the α-algorithm for inducing Petri nets from data and identifies for which class of models the approach is guaranteed to work. The α-algorithm is based on the discovery of binary relations in the log, such as the "follows" relation.

[4] is a recent work where a process model is induced in the form of a disjunction of special graphs called workflow schemes.

We differ from all of these works in three respects. First, we learn from positive and negative traces, rather than from positive traces only. Second, we use a representation that is declarative rather than procedural as Petri nets are, without sacrificing expressivity. For example we can model concurrency and synchronization among activities. Third, we can take into account attributes of events, such as in the auction protocol where we check that deadlines are respected. Wi

Other works deal with the learning of integrity constraints, in particular [10, 6, 17]. However, all of these works learn integrity constraints in the form of clauses, that are less expressive than our formalism.

6 Conclusions and Future Works

We have presented an approach for performing Process Mining by using ILP techniques. The approach introduces a new language that extends the one of disjunctive clauses and that can be used to test the compliance of a trace by simply using a Prolog interpreter. A subsumption relation for the new language is introduced together with a refinement operator.

The similarity with clausal logic allows the use of the ICL algorithm for learning process models. Two experiments have been performed on synthetic data generated from two models of interaction protocols: a sealed bid auction and the NetBill protocol. A good accuracy has been achieved in both experiments. The accuracy on NetBill is higher than the one of the α-algorithm on the same dataset.

In the future, we plan to test the system on real world process logs in order to have a more accurate test of the effectiveness of the approach.

7 Acknowledgements

This work has been partially supported by the PRIN 2005 project “Specification and verification of agent interaction protocols” and by the FIRB project “TOCAI.IT”.

References

1. 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
2. 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
3. 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
4. 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
5. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: An abductive interpretation for open societies. In Cappelli, A., Turini, F., eds.: Proceedings of the 8th Congress of the Italian Association for Artificial Intelligence (AI*IA 2003). Volume 2829 of LNAI., Springer Verlag (2003)
6. De Raedt, L., Van Laer, W.: Inductive constraint logic. In: Proceedings of the 6th Conference on Algorithmic Learning Theory. Volume 997 of LNAI., Springer Verlag (1995)
7. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In Bravetti, M., Núñez, M., Zavattaro, G., eds.: Proceedings of the Third International Workshop on Web Services and Formal Methods (WS-FM 2006). Volume 4184 of LNCS., Springer (2006)
8. Clark, K.L.: Negation as failure. In: Logic and Databases. Plenum Press (1978)
9. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: the SCIFF framework. ACM Transactions on Computational Logics (2007) Accepted for publication.
10. Raedt, L.D., Dehaspe, L.: Clausal discovery. Machine Learning **26**(2-3) (1997) 99–146
11. Apt, K.R., Bezem, M.: Acyclic programs. New Generation Comput. **9**(3/4) (1991) 335–364

12. 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
13. Cox, B., Tygar, J., Sirbu, M.: Netbill security and transaction protocol. In: Proceedings of the First USENIX Workshop on Electronic Commerce, New York (July 1995)
14. : Socs protocol repository Available at: http://edu59.deis.unibo.it:8079/SOCSProtocolsRepository/jsp/index.jsp.
15. 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
16. : Prom framework Available at: http://is.tm.tue.nl/∼cgunther/dev/prom/.
17. Jorge, A., Brazdil, P.: Integrity constraints in ilp using a monte carlo approach. In: 6th International Workshop on Inductive Logic Programming. Volume 1314 of Lecture Notes in Computer Science., Springer (1996) 229–244