Inducing Declarative Logic-Based Models from Labeled Traces

Evelina Lamma[1], Paola Mello[2], Marco Montali[2],
Fabrizio Riguzzi[1], and Sergio Storari[1]

[1] ENDIF – Università di Ferrara
Via Saragat, 1 – 44100 – Ferrara, Italy
{evelina.lamma, sergio.storari, fabrizio.riguzzi}@unife.it
[2] DEIS – Università di Bologna
viale Risorgimento, 2 – 40136 – Bologna, Italy
{pmello, mmontali}@deis.unibo.it

Abstract. In this work we propose an approach for the automatic discovery of logic-based models starting from a set of process execution traces. The approach is based on a modified Inductive Logic Programming algorithm, capable of learning a set of declarative rules.

The advantage of using a declarative description is twofold. First, the process is represented in an intuitive and easily readable way; second, a family of proof procedures associated to the chosen language can be used to support the monitoring and management of processes (conformance testing, properties verification and interoperability checking, in particular).

The approach consists in first learning integrity constraints expressed as logical formulas and then translating them into a declarative graphical language named DecSerFlow.

We demonstrate the viability of the approach by applying it to a real dataset from a health case process and to an artificial dataset from an e-commerce protocol.

Topics: Process mining, Process verification and validation, Logic Programming, DecSerFlow, Careflow.

1 Introduction

In recent years, many different proposals have been developed for mining process models from execution traces (e.g. [1,18,9]). All these approaches aim at discovering complex and procedural process models, and differ by the common structural patterns they are able to mine. While recognizing the extreme importance of such approaches, we advocate the necessity of discovering also declarative logic-based knowledge, in the form of process fragments or business rules/policies, from execution traces.

By following this approach, we do not mine a complete process model, but rather discover a set of common declarative patterns and constraints. Being

G. Alonso, P. Dadam, and M. Rosemann (Eds.): BPM 2007, LNCS 4714, pp. 344–359, 2007.

declarative, this information captures what is the high-level process behavior without expressing how it is procedurally executed, hence giving a concise and easily interpretable feedback to the business manager. The importance of adopting a declarative approach rather than an imperative one to model service flows and, more generally, business processes, has been recently pointed out in very interesting and promising works and proposals, such as ConDec [16] and DecSerFlow [17].

In this work we propose an approach for the automatic discovery of rule-based declarative models starting from a set of process execution traces, previously labeled as compliant or not. Learning a process model from both compliant and non compliant 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.

As the target language, we choose $\mathcal{S}$CIFF [4,3], a declarative language based on computational logic and abductive logic programming in particular, which was originally developed for the specification and verification of global interaction protocols. $\mathcal{S}$CIFF models interaction patterns as forward rules which state what is expected to be performed when a given condition, expressed in terms of already performed activities, holds.

An important advantage of adopting a logic programming representation is that it is possible to exploit the techniques developed in the field of Inductive Logic Programming (ILP for short) [12] for learning models from examples and background knowledge; in fact, the system ICL [8] has been adapted to the problem of learning $\mathcal{S}$CIFF constraints [11].

There are two reasons for using a $\mathcal{S}$CIFF description. First, the process is represented in an intuitive and easily readable way; second, a family of proof procedures associated to $\mathcal{S}$CIFF can be used to support the monitoring and management of processes [2] (conformance testing, properties verification and interoperability checking in particular).

Moreover, we present an approach for translating the learned $\mathcal{S}$CIFF description into a DecSerFlow/ConDec model. We call the resulting system DecMiner.

We demonstrate the viability of the approach by applying it to a real dataset from a health care process and to an artificial dataset from an e-commerce protocol.

The paper is organized as follows. Section 2 briefly introduces the $\mathcal{S}$CIFF framework. Section 3 is devoted to presenting preliminaries on ILP, on the ICL algorithm and on how it can be used to learn $\mathcal{S}$CIFF constraints. Section 4 introduces the basic concepts of DecSerFlow and shows how the learned $\mathcal{S}$CIFF constraints can be interpreted as a DecSerFlow model. Section 5 describes the experiments performed for validating the approach. Section 6 presents related works and, finally, Section 7 concludes the paper and presents directions for future work.

2 An Overview of the SCIFF Framework

The $\mathcal{S}$CIFF framework [4,3] was originally developed for the specification and verification of agent interaction protocols within open and heterogeneous societies. The framework is based on abduction, a reasoning paradigm which allows to formulate hypotheses (called *abducibles*) accounting for observations. In most abductive frameworks, *integrity constraints* are imposed over possible hypotheses in order to prevent inconsistent explanations. $\mathcal{S}$CIFF considers a set of interacting peers as an open society, formalizing interaction protocols by means of a set of global rules which constrain the external and observable behaviour of participants (for this reason, global rules are called Social Integrity Constraints).

To represent that an event ev happened (i.e., an atomic activity has been executed) at a certain time T, $\mathcal{S}$CIFF uses the symbol $\mathbf{H}(ev, T)$, where ev is a term and T is a variable. Hence, an execution trace is modeled as a set of happened events. For example, we could formalize that *bob* has performed activity a at time 5 as follows: $\mathbf{H}(a(bob), 5)$. Furthermore, $\mathcal{S}$CIFF introduces the concept of expectation, which plays a key role when defining global interaction protocols, choreographies, and more in general event-driven process. It is quite natural, in fact, to think of a process in terms of rules of the form: "if A happened, then B is expected to happen". Positive (resp. negative) expectations are denoted by $\mathbf{E}(ev, T)$ (resp. $\mathbf{EN}(ev, T)$), meaning that ev is expected (resp. not expected) to happen at time T. To satisfy a positive (resp. negative) expectation an execution trace must contain (resp. not contain) a matching happened event.

Social Integrity Constraints (ICs for short) are forward rules of the form $body \rightarrow head$, where *body* can contain literals and happened events, and *head* contains a disjunction of conjunctions of expectations and literals.

In this paper, we consider a syntax of ICs that is a subset of the one in [4,3]. In this simplified syntax, a Social Integrity Constraint, C, is a logical formula of the form

$$Body \rightarrow DisjE_1 \vee \ldots \vee DisjE_n \vee DisjEN_1 \vee \ldots \vee DisjEN_m \qquad (1)$$

We will use $Body(C)$ to indicate $Body$ and $Head(C)$ to indicate $DisjE_1 \vee \ldots \vee DisjE_n \vee DisjEN_1 \vee \ldots \vee DisjEN_m$. $Body$ is of the form $b_1 \wedge \ldots \wedge b_l$ where the b_i are literals. Some of the literals may be of the form $\mathbf{H}(ev, T)$ meaning that event ev has happened at time T.

$DisjE_j$ is a formula of the form $\mathbf{E}(ev, T) \wedge d_1 \wedge \ldots \wedge d_k$ where ev is an event and d_i are literals. All the formulas $DisjE_j$ in $Head(C)$ will be called *positive disjuncts*.

$DisjEN_j$ is a formula of the form $\mathbf{EN}(ev, T) \wedge d_1 \wedge \ldots \wedge d_k$ where ev is an event and d_i are literals. All the formulas $DisjEN_j$ in $Head(C)$ will be called *negative disjuncts*.

The literals b_i and d_i refer to predicates defined in a $\mathcal{S}$CIFF knowledge base. Variables in common to $Body(C)$ and $Head(C)$ are universally quantified ($\forall$) with scope the whole IC. Variables occurring only in $DisjE_j$ literals are existentially quantified ($\exists$) with scope the $DisjE_j$ literal itself. Variables occurring

only in $DisjEN_j$ literals are universally quantified ($\forall$) with scope the $DisjEN_j$ literal itself. An example of an IC is

$$\begin{aligned} &\mathbf{H}(a(bob), T) \wedge T < 10 \\ \rightarrow &\mathbf{E}(b(alice), T1) \wedge T < T1 \vee \\ &\mathbf{EN}(c(mary), T1) \wedge T < T1 \wedge T1 < T + 10 \end{aligned} \tag{2}$$

The interpretation of an IC is the following: if there exists a substitution of variables such that the body is true in an interpretation representing a trace, then one of the disjuncts in the head must be true. A disjunct of the form $DisjE$ means that we expect event ev to happen with T and its variables satisfying $d_1 \wedge \ldots \wedge d_k$. Therefore $DisjE$ is true if there exist a substitution of variables occurring in $DisjE$ such that ev is present in the trace.

A disjunct of the form $DisjEN$ means that we expect event ev not to happen with T and its variables satisfying $d_1 \wedge \ldots \wedge d_k$. Therefore $DisjEN$ is true if for all substitutions of variables occurring in $DisjEN$ and not appearing in $Body$ either ev does not happen or, if it happens, its properties violate $d_1 \wedge \ldots \wedge d_k$.

The meaning of the IC (2) is the following: if *bob* has executed action a at a time $T < 10$, then we expect *alice* to execute action b at some time $T1$ later than T ($\exists T1$) or we expect that *mary* does not execute action c at any time $T1$ ($\forall T1$) within 9 time units after T.

3 Learning Models

This work starts from the idea that there is a similarity between learning a $\mathcal{S}$CIFF theory, composed by a set of Social Integrity Constraints, and learning a clausal theory as described in the learning from interpretation setting of Inductive Logic Programming [12]. In fact, as a $\mathcal{S}$CIFF theory, a clausal theory can be used to classify a set of atoms (i.e. an interpretation) by returning positive unless there is at least one clause that is false in the interpretation.

A clause C is a formula in the form $b_1 \wedge \cdots \wedge b_n \rightarrow h_1 \vee \cdots \vee h_m$ where b_i are logical literals and h_i are logical atoms. A formula is ground if it does not contain variables. An interpretation is a set of ground atoms. Let us define $head(C) = \{h_1, \ldots, h_m\}$ and $body(C) = \{b_1, \ldots, b_n\}$. Sometimes we will interpret clause C as the set of literals $\{h_1, \ldots, h_m, \neg b_1, \ldots, \neg b_n\}$.

The clause C is true in an interpretation I iff, for all the substitutions θ grounding C, $(I \models body(C)\theta) \rightarrow (head(C)\theta \cap I \neq \emptyset)$. Otherwise, it is false.

Sometimes we may be given a background knowledge B with which we can enlarge each interpretation I by considering, instead of simply I, the interpretation given by $M(B \cup I)$ where M stands for a model, such as Clark's completion [7]. By using a background knowledge we are able 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 learning from interpretation setting of ILP is concerned with the following problem:

Given

- a space of possible clausal theories $\mathcal{H}$;
- a set P of positive interpretations;
- a set N of negative interpretations;
- a definite clause background theory B.

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

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

Given a disjunctive clause C (theory H) we say that C (H) *covers* the interpretation I iff C (H) is true in $M(B \cup I)$. We say that C (H) *rules out* an interpretation I iff C (H) does not cover I.

An algorithm that solves the above problem is ICL [8]. It performs a covering loop (function Learn, Figure 1) in which negative interpretations are progressively ruled out and removed from the set N. 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 N is empty or when no clause is found.

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

function FindBestClause(P, N, 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 of 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

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 in order to make it more general.

The generality order that is used is the θ-subsumption order: C is more general than D (written $C \geq D$) if there exist a substitution θ such that $D\theta \subseteq C$. If $C \geq D$ then the set of interpretation where C is true is a superset of those where D is true. The same is true if $D \subseteq C$. Thus the clauses in the beam can be gradually refined by adding literals to the body and atoms to the head. For example, let us consider the following clauses:

$C = accept(X) \vee refusal(X) \leftarrow invitation(X)$
$D = accept(X) \vee refusal(X) \leftarrow true$
$E = accept(X) \leftarrow invitation(X)$

Then C is more general than D and E, while D and E are not comparable.

The aim of FindBestClause is to discover a clause 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: dlab and rmode (see [10] for details). Given a language bias which prescribes that the body literals must be chosen among $\{invitation(X), paptest(X)\}$ 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 cannot produce a value of the heuristic function higher than that of the best clause (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 N.

3.1 Application of ICL to Integrity Constraint Learning

An approach to applying ICL for learning ICs is described in [11]. Each IC is seen as a clause that must be true in all the positive interpretations (compliant execution traces) and false in some negative interpretation (non compliant execution traces). The theory composed of all the ICs must be such that all the ICs are true when considering a compliant trace and at least one IC is false when considering a non compliant one.

In order to apply ICL, a generality order and a refinement operator for ICs must be defined. The generality order is the following: an IC C is more general than an IC D (written $C \geq D$) if there exists a substitution θ for the variables of $body(D)$ such that $body(D)\theta \subseteq body(C)$ and, for each disjunct d in the head of D:

- if d is positive, then there exist a positive disjunct c in the head of C such that $d\theta \supseteq c$
- if d is negative, then there exist a negative disjunct c in the head of C such that $d\theta \subseteq c$

For example, the IC

$$\mathbf{H}(invitation, T) \wedge \mathbf{H}(accept, T3) \rightarrow \mathbf{E}(papTest, T1) \wedge T1 > T \vee \mathbf{E}(refusal, T2) \wedge T2 > T$$

is more general than

$$\mathbf{H}(invitation, T) \wedge \mathbf{H}(accept, T3) \rightarrow \mathbf{E}(papTest, T1) \wedge T1 > T$$

which in turn is more general than

$$\mathbf{H}(invitation, T) \rightarrow \mathbf{E}(papTest, T1) \wedge T1 > T$$

Moreover

$$\mathbf{H}(invitation, T) \rightarrow \mathbf{E}(papTest, T1) \vee \mathbf{E}(refusal, T2)$$

is more general than

$$\mathbf{H}(invitation, T) \rightarrow \mathbf{E}(papTest, T1) \vee \mathbf{E}(refusal, T2) \wedge T2 > T$$

and

$$\mathbf{H}(sendPapTestResult(neg), T) \rightarrow \mathbf{EN}(papTest, T1) \wedge T1 > T$$

is more general than

$$\mathbf{H}(sendPapTestResult(neg), T) \rightarrow \mathbf{EN}(papTest, T1)$$

A refinement operator can be obtained in the following way: given an IC C, obtain a refinement D by:

- adding a literal to the body;
- adding a disjunct to the head;
- removing a literal from a positive disjunct in the head;
- adding a literal to a negative disjunct in the head.

The language bias specifies which literals can be added to the body, which disjuncts can be added to the head and which literals can be added or removed from head disjuncts.

When adding a disjunct to the head, the refinement operator behaves differently depending on the sign of the disjunct:

– in the case of a positive disjunct, the disjunct formed by the **E** literal plus all the literals in the language bias for the disjunct is added;
– in the case of a negative disjunct, only the **EN** literal is added.

Given an IC C, the refinement operator returns a set of ICs $\rho(C)$ that contains the ICs obtained by applying in all possible ways one of the above mentioned operations. Every IC of $\rho(C)$ is more general than C.

4 From $\mathcal{S}$CIFF Integrity Constraints to DecSerFlow

The meaning of the learned $\mathcal{S}$CIFF Integrity Constraints is very close to the one of various DecSerFlow relation formulas [17]. We therefore tackled the problem of translating a DecSerFlow model into a set of ICs and vice-versa, with the aim of integrating the advantages of both approaches:

– DecSerFlow represents a process model in a declarative and user-friendly graphical notation;
– $\mathcal{S}$CIFF Integrity Constraints are declarative intuitive rules easy to read by humans;
– DecSerFlow has a mapping to LTL and hence could be used to perform monitoring functionalities or to directly enact the model;
– the $\mathcal{S}$CIFF framework associates to the $\mathcal{S}$CIFF language a family of proof procedures capable of performing conformance testing, properties verification and interoperability checking.

DecSerFlow is briefly described in Section 4.1 giving an intuition about the translation from a DecSerFlow model to the $\mathcal{S}$CIFF formalism as addressed in [6]. We then describe in Section 4.2 how we can learn DecSerFlow constraints from labeled traces.

4.1 DecSerFlow: A Brief Recap

DecSerFlow is a graphical language that adopts a declarative style of modeling: the user does not specify possible process flows but only a set of constraints (namely policies or business rules) among activities. For a detailed description of the language and its mapping to Linear Temporal Logic, see [17].

To illustrate the advantages of declarative modeling, the authors consider the problem of specifying that two different activities should not be executed together (i.e. it is possible to execute the first or the latter activity multiple times, but the two activities exclude each other). A procedural language is not able to directly represent the requirement and must explicitly represent all the possible executions (see Figure 2), leading to some problems:

– the process becomes over-specified;
– the modeler must introduce decision points to handle the possible executions, but it is not clear how and when these decisions should be evaluated.

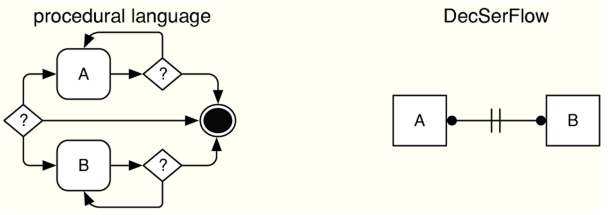

Fig. 2. Procedural vs. declarative approach when modeling the not coexistence between two activities

Instead, by using a declarative language such as DecSerFlow, forbidding the coexistence of two activities A and B may be expressed by a special edge between the two nodes representing A and B. This will be translated into the simple LTL formula: $\neg(\diamond A \wedge \diamond B)$.

As shown in Figure 2, the basic intuitive concepts of DecSerFlow are: *activities*, atomic units of work; *constraints among activities*, to model policies/business rules and constrain their execution.

Constraints are given as relationships between two (or more) activities. Each constraint is then expressed as an LTL formula, hence the name "formulas" to indicate DecSerFlow relationships.

DecSerFlow core relationships are grouped into three families:

- *existence formulas*, unary relationships used to constrain the cardinality of activities
- *relation formulas*, which define (positive) relationships and dependencies between two (or more) activities;
- *negation formulas*, the negated version of relation formulas (as in $\mathcal{S}$CIFF, DecSerFlow follows an open approach i.e. the model should express not only what has to be done but also what is forbidden).

The intended meaning of DecSerFlow formulas can be expressed by using $\mathcal{S}$CIFF. In [6], the authors propose a translation by mapping atomic DecSerFlow activities to $\mathcal{S}$CIFF events and formulas to corresponding integrity constraints.

For example, let us consider the *succession* formula among two whatsoever activities A and B: it states that every execution of A should be *followed* by the execution of B and each B should be *preceded* by A, i.e. that B is *response* of A and, in turn, A is *precedence* of B. This formula could be translated as follows. First, the succession between activities is mapped to the response and precedence formulas, as described above; the response and precedence formulas are then both formalized by using a specific integrity constraint. In particular, the response formula between A and B is mapped to

$$\mathbf{H}(A, T_A) \rightarrow \mathbf{E}(B, T_B) \wedge T_B > T_A. \tag{3}$$

while the precedence formula between B and A is mapped to

$$\mathbf{H}(B, T_B) \rightarrow \mathbf{E}(A, T_A) \wedge T_A < T_B. \tag{4}$$

4.2 Learning DecSerFlow Models

In order to learn DecSerFlow models, we first learn $\mathcal{S}$CIFF ICs and then manually translate them into DecSerFlow constraints using the equivalences discussed in the previous section. We call the system implementing this approach DecMiner.

We decided to use $\mathcal{S}$CIFF as intermediate language instead of LTL because it can handle times and data in an explicit and quantitative way, exploiting Constraint Logic Programming to define temporal and data-related constraints. This is useful to deal with many processes as, for example, the Screening and NetBill ones described in details in Section 5. Moreover it allows to think about how to extend DecSerFlow to explicitly consider time and data. However, at the moment, $\mathcal{S}$CIFF does not support model enactment and we are working on an extension of the $\mathcal{S}$CIFF proof procedure capable of dealing with it.

To ease the translation, we provide ICL with a language bias ensuring that the learned ICs can be translated into DecSerFlow.

Thus the language bias takes the form of a set of templates that are couples (BS, HS): BS is a set that contains the literals that can be added to the body and HS is a set that contains the disjuncts that can be added to the head. Each element of HB is a couple $(Sign, Literals)$ where $Sign$ is either + for a positive disjunct or - for a negative disjunct, and $Literals$ contains the literals that can appear in the disjunct. We will have a set of templates for each DecSerFlow constraint, where each template in the set is an application of the constraint to a set of activities.

5 Experiments

The experiments have been performed over a real dataset and an artificial dataset. The real dataset regards a health care process while the artificial dataset regards an e-commerce protocol.

5.1 Cervical Cancer Screening Careflow and Log

As a case study for exploiting the potentialities of our approach we choose the process of cervical cancer screening [5] proposed by the sanitary organization of the Emilia Romagna region of Italy. Cervical cancer is a disease in which malignant (cancer) cells form in the tissues of the cervix of the uterus. The screening program proposes several tests in order to early detect and treat cervical cancer. It is usually composed by five phases: Screening planning; Invitation management; First level test with pap-test; Second level test with colposcopy, and eventually biopsy. The process is composed by 16 activities.

To perform our experiments we collected 157 traces from a database of an Italian cervical cancer screening center. All the 157 traces have been analyzed by a domain expert and labeled as compliant or non compliant with respect to the cervical cancer screening protocol adopted by the screening center. The traces classified as compliant were 55 over 157.

Each event trace was then adapted to the format required by the ICL algorithm, transforming each trace into an interpretation. For this preliminary study, we considered only the performed activities (without taking into account originators and other parameters, except from the posted examinations results); furthermore, we use sequence numbers rather than actual execution times.

An example of an interpretation is the following:

```
begin(model(m1)).
H(invitation,1).
H(refusal,2).
end(model(m1)).
```

5.2 NetBill

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. It is composed of two phases: negotiation and transaction. In the negotiation phase, the customer requests a price for a good from the merchant, the merchant propose a price for the good and the customer can accept the offer, refuse it or make another request to the merchant, thus initiating a new negotiation. The transaction phase starts if the customer accepts the offer: the merchant delivers the good to the customer encrypted with key K; the customer creates an electronic purchase order (EPO) that is countersigned by the merchant that add also the value of K and send the EPO to the NetBill server; the NetBill server checks the EPO and if customer's account contains enough funds it transfers the price to the merchant's account and sends a signed receipt that includes the value K to the merchant; 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 [13]. One of them is

$$\begin{aligned} & \mathbf{H}(request(C, M, good(G, Q), Nneg, Trq)) \wedge \\ & \mathbf{H}(present(M, C, good(G, Q), Nneg, Tp)) \wedge Trq \leq Tp \\ \rightarrow & \mathbf{E}(accept(C, M, good(G, Q)), Ta) \wedge Tp \leq Ta \vee \\ & \mathbf{E}(refuse(C, M, good(G, Q)), Trf) \wedge Tp \leq Trf \vee \\ & \mathbf{E}(request(C, M, good(G, Qrql), Nnegl), Trql) \wedge Tp \leq Trql \end{aligned} \tag{5}$$

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*.

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 compliant or non compliant traces depending on the result of the test. The process is repeated until 2000 compliant traces and 2000 non compliant traces have been generated.

5.3 Results

Five experiments have been performed for the screening and the NetBill processes. For the screening process, five folds cross validation was used, i.e., the dataset was divided into 5 sets and in each experiment 4 were used for training and the remaining for testing. For NetBill, the training and testing set were generated with the procedure sketched above with different seeds for the random function for each experiments.

DecMiner, the α-algorithm [19] and the Multi-Phase Mining approach [20] have been applied in each experiment. The α-algorithm is one of the first process mining algorithms and it induces Petri nets. The Multi-Phase (MP) mining algorithm can be used to construct an Event-driven Process Chain (EPC) that can be also translated in a Petri net. We used the implementation of these algorithms available in the ProM suite [14]. Since the α-algorithm and the Multi-Phase miner take as input a single set of traces, we have provided them with the compliant traces only.

Table 1. Results of the experiments

Experiment	DecMiner	α algorithm	MP algorithm
Screening	97.44%	96.15%	94.89%
NetBill	96.11%	66.81%	60.52%

The average accuracy of each algorithm is reported in Table 1. The accuracy is defined as the number of compliant traces that are correctly classified as compliant by the learned model plus the number of non compliant traces that are correctly classified as not compliant by the learned model divided by the total number of traces. Compliance of an execution trace with respect to a learned Petri net has been evaluated by using the *Conformance checker* ProM plug-in.

The average time taken by DecMiner are 2 minutes for Screening and 6.5 hours for NetBill on a Athlon XP64 1.80 GHz machine. The average times taken by the α-algorithm and the MP miner are under one minute for both datasets.

5.4 Mapping the Learned ICs to DecSerFlow

In order to illustrate the behavior of the approach for inducing DecSerFlow constraints, we report in this Section the ICs learned from the screening dataset together with their translation into DecSerFlow constraints.

Running DecMiner on the screening dataset, we obtained the following ICs:

$$\begin{aligned} &true \\ \rightarrow &\mathbf{E}(examExecution(papTest), A) \ \vee \mathbf{E}(refusal, B) \end{aligned} \qquad (IC_1)$$

(IC_1) states that there must be a pap test execution or a refusal.

$$\begin{aligned} &\mathbf{H}(resultPosting(positive, papTest), A) \\ \rightarrow &\mathbf{E}(examExecution(colposcopy), B) \end{aligned} \qquad (IC_2)$$

(IC_2) states that if there is a positive pap test then there must be also a colposcopy.

$$\begin{aligned} &\mathbf{H}(examExecution(papTest), A) \\ \rightarrow &\mathbf{E}(invitation, B) \wedge prec(B, A) \end{aligned} \qquad (IC_3)$$

(IC_3) states that a pap-test execution must be immediately preceeded by an invitation.

$$\begin{aligned} &\mathbf{H}(resultPosting(doubtful, colposcopy), A) \\ \rightarrow &\mathbf{E}(examExecution(biopsy, B) \wedge less(A, B) \end{aligned} \qquad (IC_4)$$

(IC_4) states that a biopsy should be executed after a doubtful colposcopy.

The predicates $less(A, B)$ and $prec(A, B)$ are defined in the background knowledge as follows:

$less(A, B) \leftarrow A < B - 1.$
$prec(A, B) \leftarrow A \ is \ B - 1.$

The ICs have been mapped into DecSerFlow constraints in the following way:

- IC_1 is translated into a mutual substitution constraint between *examExecution(papTest)* and *refusal*.
- IC_2 is translated into a responded presence constraint between *resultPosting(positive,papTest)* and *examExecution(colposcopy)*.
- IC_3 is translated into a chain precedence constraint between *invitation* and *examExecution(papTest)* meaning that *examExecution(papTest)* must be immediately preceeded by *invitation*.
- IC_4 is translated into a response constraint between $resultPosting(doubtful, colposcopy)$ and $examExecution(biopsy)$.

The resulting DecSerFlow model is shown in Figure 3.

In the future we plan to automate this translation process. This will require an appropriate tuning of the language bias in order to learn constraints very close to the form of the template constraints used in [6].

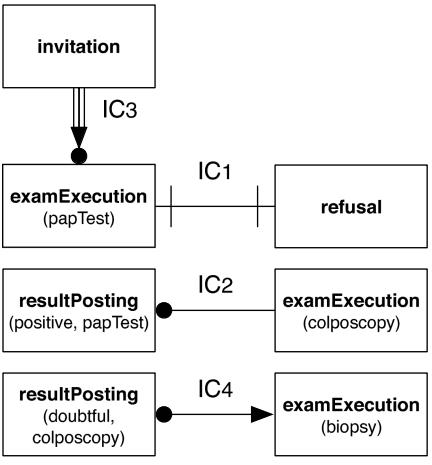

Fig. 3. The DecSerFlow representation of the ICs learned from the event log

6 Related Works

[1] 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.

[19] 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.

In [20] the authors describe an algorithm which derives causal dependencies between activities and use them for constructing instance graphs, presented in terms of Event-driven Process Chains (EPCs).

[9] 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 compliant and non compliant traces, rather than from compliant traces only. Second, we use a representation that is declarative rather than procedural as Petri nets are, without sacrificing expressivity. For example, the $\mathcal{S}$CIFF language supports conjunction of happened events in the body, to model complex triggering conditions, as well as disjunctive expectations in the head. Third, our language is able to model and reason upon data, by exploiting either the underlying Constraints Solver or the Prolog inference engine.

In [15] the authors use an extension of the Event Calculus (EC) of Kowalski and Sergot to declaratively model event based requirements specifications. The choice of EC is motivated by both practical and formal needs, that are shared by our approach. In particular, in contrast to pure state-transition representations, both the EC and $\mathcal{S}$CIFF representations include an explicit time structure and are very close to most event-based specifications. Moreover they allows us to use the same logical foundation for verification at both design time and runtime [2].

In this paper, however, our emphasis is about the learning of the model, instead of the verification issue. We deal with time by using suitable CLP constraints on finite domains, while they use a temporal formalism based on Event Calculus.

We are aware that the temporal framework we use is less expressive than EC, but we think that it is enough powerful for our goals and is a good trade off between expressiveness and efficiency.

7 Conclusions

In this work we presented the result of our research activity aimed at proposing a methodology for analyzing a log containing several traces labeled as compliant or not compliant. From them we learn a $\mathcal{S}$CIFF theory, containing a minimal set of constraints able to accurately classify a new trace.

The proposed methodology is based on Inductive Logic Programming and, in particular, on the ICL algorithm. Such an algorithm is adapted to the problem of learning integrity constraints in the $\mathcal{S}$CIFF language. By considering not only compliant traces, but also non compliant ones, our approach is able to learn a model which expresses not only what should be done, but also what is forbidden.

Furthermore, the learned $\mathcal{S}$CIFF formulas can be translated into DecSerFlow constraints. We called the resulting system DecMiner.

In order to test the proposed methodology, we performed a number of experiments on two dataset: a cervical cancer screening log and an e-commerce log. The accuracy of DecMiner was compared with the one of the α-algorithm. Moreover, the ICs learned from the screening dataset are shown together with their translation into DecSerFlow.

In the future, we plan to make the translation from the $\mathcal{S}$CIFF formalism into the DecSerFlow one automatic. Moreover, we plan to consider explicitly activity originators and the actual execution time of each event (for example represented as the number of days from the 1st of January 1970) in order to learn constraints which involve also deadlines. Finally, we will investigate the effect of noise on DecMiner, by studying the effect of misclassified examples.

Acknowledgments. This work has been partially supported by NOEMALIFE under the "SPRING" regional PRRITT project, by the PRIN 2005 project "Specification and verification of agent interaction protocols" and by the FIRB project "TOCAI.IT".

References

1. Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 469–483. Springer, Heidelberg (1998)
2. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Storari, S., Torroni, P.: Computational logic for run-time verification of web services choreographies: Exploiting the ocs-si tool. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 58–72. Springer, Heidelberg (2006)

3. 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 (accepted for publication, 2007)
4. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: An abductive interpretation for open societies. In: Cappelli, A., Turini, F. (eds.) AI*IA 2003. LNCS, vol. 2829, Springer, Heidelberg (2003)
5. Cervical cancer screening web site: Available at: http://www.cancer.gov/cancertopics/pdq/screening/cervical/healthprofessional
6. Chesani, F., Mello, P., Montali, M., Storari, S.: Towards a decserflow declarative semantics based on computational logic. Technical Report DEIS-LIA-07-002, DEIS, Bologna, Italy (2007)
7. Clark, K.L.: Negation as failure. In: Logic and Databases, Plenum Press, New York (1978)
8. De Raedt, L., Van Laer, W.: Inductive constraint logic. In: Zeugmann, T., Shinohara, T., Jantke, K.P. (eds.) ALT 1995. LNCS, vol. 997, Springer, Heidelberg (1995)
9. Greco, G., Guzzo, A., Pontieri, L., Saccá, D.: Discovering expressive process models by clustering log traces. IEEE Trans. Knowl. Data Eng. 18(8), 1010–1027 (2006)
10. ICL manual: Available at: http://www.cs.kuleuven.be/~ml/ACE/Doc/ACEuser.pdf
11. Lamma, E., Mello, P., Riguzzi, F., Storari, S.: Applying inductive logic programming to process mining. In: ILP 2007, Springer, Heidelberg (2007)
12. Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. Journal of Logic Programming 19/20, 629–679 (1994)
13. SCIFF specification of the netbill protocol: Available at: http://edu59.deis.unibo.it:8079/SOCSProtocolsRepository/jsp/protocol.jsp?id=8
14. Prom framework: Available at: http://is.tm.tue.nl/~cgunther/dev/prom/
15. Rouached, M., Perrin, O., Godart, C.: Towards formal verification of web service composition. In: Dustdar, S., Fiadeiro, J.L., Sheth, A. (eds.) BPM 2006. LNCS, vol. 4102, pp. 257–273. Springer, Heidelberg (2006)
16. van der Aalst, W.M.P., Pesic, M.: A declarative approach for flexible business processes management. In: Eder, J., Dustdar, S. (eds.) Business Process Management Workshops. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)
17. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 1–23. Springer, Heidelberg (2006)
18. 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), 237–267 (2003)
19. 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), 1128–1142 (2004)
20. van Dongen, B.F., van der Aalst, W.M.P.: Multi-phase process mining: Building instance graphs. In: Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T.-W. (eds.) ER 2004. LNCS, vol. 3288, pp. 362–376. Springer, Heidelberg (2004)