Exploiting Inductive Logic Programming Techniques for Declarative Process Mining

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

[1] DEIS – Università di Bologna
viale Risorgimento, 2 – 40136 – Bologna, Italy
{federico.chesani,paola.mello,marco.montali}@unibo.it
[2] ENDIF – Università di Ferrara
Via Saragat, 1 – 44100 – Ferrara, Italy
{evelina.lamma,fabrizio.riguzzi,sergio.storari}@unife.it

Abstract. In the last few years, there has been a growing interest in the adoption of declarative paradigms for modeling and verifying process models. These paradigms provide an abstract and human understandable way of specifying constraints that must hold among activities executions rather than focusing on a specific procedural solution. Mining such declarative descriptions is still an open challenge. In this paper, we present a logic-based approach for tackling this problem. It relies on Inductive Logic Programming techniques and, in particular, on a modified version of the Inductive Constraint Logic algorithm. We investigate how, by properly tuning the learning algorithm, the approach can be adopted to mine models expressed in the ConDec notation, a graphical language for the declarative specification of business processes. Then, we sketch how such a mining framework has been concretely implemented as a ProM plug-in called DecMiner. We finally discuss the effectiveness of the approach by means of an example which shows the ability of the language to model concurrent activities and of DecMiner to learn such a model.

1 Introduction

When facing the problem of defining and developing a Business Process (BP), we can mainly identify two different and complementary roles: the *business analyst*, a domain expert aiming at improving the performances of her company, and the *IT-expert*, who has the responsibility of bringing business-level models to an effective underlying implementation. The complementarity of these roles leads to different perspectives about the process to be developed: while the IT-expert typically adopts a procedural style of modeling, dealing with implementation aspects and trying to obtain an executable process, the business analyst follows a more declarative approach (see Figure 1). Indeed, at a business level it is very important to represent in an intuitive and concise way the domain and problem under study, rather than focusing on a specific solution. In this respect, the

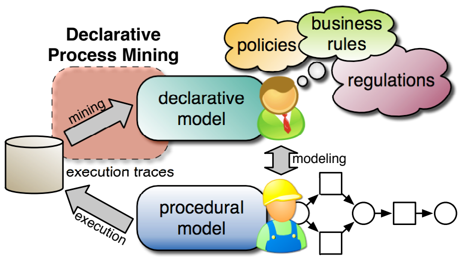


Fig. 1. Declarative and procedural perspectives when modeling Business Processes

model will typically involve business rules, covering best practices and internal constraints as well as internal/external regulations and compliance requirements.

The importance of adopting a declarative style of modeling has been recently pointed out by van der Aalst and Pesic [18]: we agree with their claim that declarative languages fit better complex, unpredictable processes, where a good balance between support and flexibility is of key importance. To this end, in [18] they propose a new graphical language for specifying process flows in a declarative manner. The language, called ConDec, does not completely fix the control flow among activities, but rather envisages 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 they are not explicitly forbidden. ConDec adopts an underlying semantics by means of Linear Temporal Logics (LTL), and can also be mapped onto a logic programming-based framework called $\mathcal{S}$CIFF (Social Constrained IFF) [2,4], which was originally developed for the specification and verification of global interaction protocols in open Multi-Agent Systems but has recently been applied in the context of BPs and SOA (Service-Oriented Architecture) Choreographies. $\mathcal{S}$CIFF provides a declarative language based on Computational Logic, where 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 in the style of Constraint Logic Programming (CLP) [12]. $\mathcal{S}$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.

An important topic related to declarative process specification, which is still an open challenge, concerns their discovery starting from execution traces, i.e., *declarative process mining*. Indeed, up to now, the goal of process mining has been the discovery of procedural process models (such as Petri Nets or Event-driven Process Chains [21,24]). We claim the necessity of mining also declarative models, to enable the possibility of inferring essential process constraints, easily understandable by business analysts and not affected by procedural details.

In this paper, we present a logic-based approach to address this issue. It relies on Inductive Logic Programming (ILP) techniques and, in particular, on a modified version of the Inductive Constraint Logic (ICL) algorithm [15]. The algorithm takes as input a set of process execution traces, previously labeled as compliant or not, and produces a set of $\mathcal{S}$CIFF rules which correctly classify them. This algorithm has been further modified, by properly tuning it and relying on the mapping presented in [4], for learning ConDec models. Then, we describe how the whole approach has been implemented as a plug-in of the ProM [23] process mining framework. The plug-in, called DecMiner, is capable of mining ConDec models starting from a set of process execution traces. The plug-in envisages different phases, ranging from the classification of traces into compliant and non-compliant subsets to the choice of which ConDec constraints have to be considered and finally to the presentation of the mined model. The effectiveness of the approach is illustrated by considering an example inspired by the one presented in [17] that involves the management of a hotel and spa.

Our previous papers on process mining [14,13] focused on the algorithm for learning $\mathcal{S}$CIFF rules and presented only a sketch of the technique for the translation into ConDec. In this work we describe how we automated this process and implemented it into the DecMiner ProM plug-in.

The paper is organized as follows. Section 2 describes the declarative languages we consider, namely $\mathcal{S}$CIFF and ConDec, and the mapping between ConDec and a subset of SCIFF rules. Section 3 presents the learning process and the DecMiner plug-in. Section 4 discusses the experiments performed for validating the approach. Section 5 presents related works and, finally, Section 6 concludes the paper and discusses future work.

2 Declarative Specification of Business Processes

In this section, we first briefly introduce the $\mathcal{S}$CIFF language, a logic-based language originally developed for specifying and verifying interaction protocols in open Multi-Agent Systems [2]. We then briefly describe ConDec [18], a graphical language supporting the intuitive modeling of declarative constraints on the flow of activities. Finally, we sketch how $\mathcal{S}$CIFF can be exploited to formalize ConDec models as well as to extend its expressiveness, relying on the results presented in [4].

2.1 An Overview of the $\mathcal{S}$CIFF Framework

The $\mathcal{S}$CIFF framework [2] 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 (constraints) which constrain the external and observable behavior of participants.

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 or a number indicating the time. Hence, an execution trace is modeled as a set of executed (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 processes. It is quite natural, in fact, to think of a process in terms of rules of the form: "if ev_1 happened, then ev_2 is expected to happen." Positive expectations are denoted by $\mathbf{E}(ev, T)$ meaning that ev is expected to happen at time T. To satisfy a positive expectation, an execution trace must contain a matching happened event. Negative expectations are denoted by $\mathbf{EN}(ev, T)$ meaning that ev is expected not to happen at time T. To satisfy a negative expectation an execution trace must not contain a matching happened event.

$\mathcal{S}$CIFF Integrity Constraints (ICs for short) are forward rules of the form *body* $\rightarrow$ *head*, where *body* can contain literals (i.e. a logical atom or its negation) 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 [2]. In this simplified syntax, an IC 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 $DisjE_1 \vee \ldots \vee DisjE_n \vee DisjEN_1 \vee \ldots \vee DisjEN_m$ of a rule C. $Body$ is of the form $b_1 \wedge \ldots \wedge b_l$ where the b_is 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 the d_is 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 the d_is are literals. All the formulas $DisjEN_j$ in $Head(C)$ will be called *negative disjuncts*.

The event ev can be a term. The literals b_is and d_is 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 positive disjuncts are existentially quantified ($\exists$) with scope the disjunct itself. Variables occurring only in negative disjuncts are universally quantified ($\forall$) with scope the disjunct itself. An example of an IC is

$$\begin{aligned}(IC.1)\quad & H(a(bob), T) \wedge T < 10 \\ & \rightarrow E(b(alice), T1) \wedge T < T1 \vee \\ & EN(c(mary), T2) \wedge T < T2 \wedge T2 < T + 10\end{aligned}$$

The meaning of the $IC.1$ 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 $T2$ ($\forall T2$) within 9 time units after T.

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 positive disjunct means that we expect event ev to happen with T and its variables satisfying $d_1 \wedge \ldots \wedge d_k$. Therefore the disjunct is true if there exist a substitution of variables occurring in it such that *ev* is present in the trace and the d_is are satisfied. A negative disjunct means that we expect event *ev* not to happen with T and its variables satisfying $d_1 \wedge \ldots \wedge d_k$. Therefore the disjunct is true if for all substitutions of variables occurring in it 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 main and original application of the $\mathcal{S}$CIFF framework and its proof procedure is to verify whether an execution of the process concretely adheres to the specification, i.e., to perform *compliance checking*. $\mathcal{S}$CIFF is seamlessly able to check compliance both at run-time, by dynamically collecting and reasoning upon occurring events, or a posteriori, by analyzing the log of an observed execution trace.

Roughly speaking, $\mathcal{S}$CIFF combines occurred events with the specified rules, to suitably generate the corresponding expectations; then expectations are verified against the execution trace: a positive expectation must have a corresponding matching event, whereas a negative expectation forbids the presence of a matching event. If such conditions are not met (i.e., a positive/negative expectation is not/is matched by a corresponding event), then the expectations are violated, and the execution trace is evaluated as non-compliant.

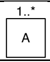
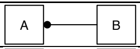
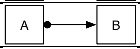
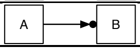
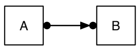
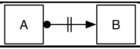
A posteriori compliance checking has been wrapped into a ProM plug-in called $\mathcal{S}$CIFFChecker [3], which can be exploited to classify MXML execution traces as compliant or non-compliant w.r.t. a high-level declarative criterion. Such a criterion is specified by configuring reactive business rules expressed in a natural language-like manner and by automatically mapping them onto the underlying formalism.

2.2 ConDec and Its $\mathcal{S}$CIFF Mapping

ConDec [18,16] is a graphical language suitable for the declarative specification of flexible Business Processes. Flexibility is provided since ConDec does not fix a completely specified process flow, but rather imposes only the (minimal) set of constraints that must be satisfied when executing the process activities. Constraints are policies/business rules which can be exploited to describe both what is mandatory and what is forbidden in the process. They are mainly organized into three basic groups: (i) *existence constraints*, unary relationships constraining the cardinality of activity executions; (ii) *relation constraints*, positive relationships between two activities used to specify what should be executed when a given situation holds; (iii) *negation constraints*, the negated version of relation ones, imposed to forbid the execution of a certain activity when a given situation holds.

We have provided a complete mapping of ConDec relationships to $\mathcal{S}$CIFF [4]. Table 1 shows some basic ConDec constraints, together with their corresponding formalization. For example, the existence constraint specifies that the involved activity must be executed at least once; this can be expressed in $\mathcal{S}$CIFF by simply stating that the activity is *expected to happen*. The responded existence between A and B imposes the existence of B only if activity A is executed, without putting any temporal condition between the two executions. Temporizing such

Table 1. Mapping of some ConDec formulas onto $\mathcal{S}$CIFF

	Name	$\mathcal{S}$CIFF Mapping
	existence	true $\rightarrow \mathbf{E}(A,T)$
	responded existence	$\mathbf{H}(A,T_A) \rightarrow \mathbf{E}(B,T_B)$
	response	$\mathbf{H}(A,T_A) \rightarrow \mathbf{E}(B,T_B) \wedge T_B > T_A$
	precedence	$\mathbf{H}(B,T_B) \rightarrow \mathbf{E}(A,T_A) \wedge T_A < T_B$
	succession	$\mathbf{H}(A,T_A) \rightarrow \mathbf{E}(B,T_B) \wedge T_B > T_A$ $\mathbf{H}(B,T_B) \rightarrow \mathbf{E}(A,T_A) \wedge T_A < T_B$
	negation response	$\mathbf{H}(A,T_A) \rightarrow \mathbf{EN}(B,T_B) \wedge T_B > T_A$

a constraint leads either to a response or precedence constraint, depending on what kind of ordering is imposed between the two activities. For example, response states that if activity A has been performed, then B must be performed afterward; the "after" ordering can be modeled in $\mathcal{S}$CIFF by putting a "greater than" CLP [12] constraint among the execution time associated to B and the one associated to A, i.e. $T_B > T_A$. The precedence constraint is modeled in a similar way, by inverting the constraint to express a "before" relationship.

Finally, ConDec supports also negative constraints, i.e., constraints used to forbid the execution of certain activities. They are mapped to $\mathcal{S}$CIFF similarly to positive relation constraints but imposing negative expectations instead of positive ones (see, for example, the negation response constraint in Table 1, which states that after activity A it is not possible to execute B anymore, being T_B universally quantified with scope the disjunct where it appears).

$\mathcal{S}$CIFF can be used not only to formalize ConDec, but also to support different extensions to the language, such as: (i) considering conjunction of events in relationships (e.g., to model synchronizing responses, namely responses which trigger only when two or more events occur); (ii) involving quantitative temporal constraints, such as deadlines and delays; (iii) constraining also data involved in the activities execution, such as originators.

2.3 Running Example

In order to explain how the declarative mining approach works, we use a process model that is inspired to [17] as a running example. This model describes a simple process of renting rooms and services in a hotel and spa. 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 later time (e.g the client may decide to use another credit card). During her stay, the client can require one or more room, laundry and massage services. Each service, identified by a code, is followed by the respective registration of the service costs into the client bill. Of course, each service cost must be registered only if the service has been effectively provided to the client and only one time. Moreover, if the client chooses a shiatzu massage, the spa presents her a special offer. The cost related to 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 eleven activities and eleven constraints. Activities *register_client_data*, *check_out* and *charge* are about the check-in/check-out of the client and expenses charging. Activities *room_service*, *laundry_service*, and *massage_service* log which services have been accessed to by the client, while billings for each service are represented by corresponding activities. For each activity, a unique identifier is introduced to correctly charge the clients with the billings for the services they effectively made use of. Moreover, for the massage related activities, an additional parameter is used to specify the massage type (aromatic or shiatzu). Finally, the activity *shiatzu_offer* maps the business policy of offering a special packet/discount to clients interested in shiatzu massages.

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 repetitions 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) *bill_massage_service* must be executed after each *massage_service*, and *bill_massage_service* can be executed only if the *massage_service* activity has been executed before;
- (C.5) *shiatzu_offer* must be executed after a *massage_service* activity with type shiatzu;
- (C.6) *check_out* must be performed in every process instance;
- (C.7) *charge* must be performed in every process instance;
- (C.8) *bill_nights* must be performed in every process instance.
- (C.9) *bill_room_service* must be executed only one time for each service identifier;
- (C.10) *bill_laundry_service* must be executed only one time for each service identifier;
- (C.11) *bill_massage_service* must be executed only one time for each service identifier;

The $\mathcal{S}$CIFF representation is composed by the following ICs:

$$
\begin{array}{rl}
(C.1) & true \\
& \rightarrow E(register\_client\_data, Trcd)) \wedge Trcd = 1. \\
(C.2) & H(room\_service(rs\_id(IDrs)), Trs) \\
& \rightarrow E(bill\_room\_service(rs\_id(IDbrs)), Tbrs) \wedge \\
& IDrs = IDbrs \wedge Tbrs > Trs. \\
& H(bill\_room\_service(rs\_id(IDbrs)), Tbrs) \\
& \rightarrow E(room\_service(rs\_id(IDrs)), Trs) \wedge \\
& IDbrs = IDrs \wedge Trs < Tbrs. \\
(C.3) & H(laundry\_service(la\_id(IDls)), Tls) \\
& \rightarrow E(bill\_laundry\_service(la\_id(IDbls)), Tbls) \wedge \\
& IDls = IDbls \wedge Tbls > Tls. \\
& H(bill\_laundry\_service(la\_id(IDbls)), Tbls) \\
& \rightarrow E(laundry\_service(la\_id(IDls)), Tls) \wedge \\
& IDbls = IDls \wedge Tls < Tbls. \\
(C.4) & H(massage\_service(ma\_id(IDms), type(TYms))), Tms) \\
& \rightarrow E(bill\_massage\_service(ma\_id(IDbms), type(TYbms)), Tbms) \wedge \\
& IDms = IDbms \wedge TYms = TYbms \wedge Tbms > Tms. \\
& H(bill\_massage\_service(ma\_id(IDbms), type(TYbms))), Tbms) \\
& \rightarrow E(massage\_service(ma\_id(IDms), type(TYms)), Tms) \wedge \\
& IDbms = IDms \wedge TYbms = TYms \wedge Tms < Tbms. \\
(C.5) & H(massage\_service(ma\_id(IDms), type(TYms)), Tms) \wedge TYms = shiatzu \\
& \rightarrow E(shiatzu\_offer, Tbms) \wedge Tbms > Tms. \\
(C.6) & true \\
& \rightarrow E(check\_out, Tco). \\
(C.7) & true \\
& \rightarrow E(charge, Tch). \\
(C.8) & true \\
& \rightarrow E(bill\_nights, Tbn). \\
(C.9) & H(bill\_room\_service(rs\_id(IDbrs1)), Tbrs1) \\
& \rightarrow EN(bill\_room\_service(rs\_id(IDbrs2)), Tbrs2) \wedge \\
& IDbrs1 = IDbrs2 \wedge Tbrs2 > Tbrs1. \\
(C.10) & H(bill\_laundry\_service(la\_id(IDbls1)), Tbls1) \\
& \rightarrow EN(bill\_laundry\_service(la\_id(IDbls2)), Tbls2) \wedge \\
& IDbls1 = IDbls2 \wedge Tbls2 > Tbls1. \\
(C.11) & H(bill\_massage\_service(ma\_id(IDbms1), type(TYbms1))), Tbms1) \\
& \rightarrow EN(bill\_massage\_service(ma\_id(IDbms2), type(TYbms2)), Tbms2) \wedge \\
& IDbms1 = IDbms2 \wedge TYbms1 = TYbms2 \wedge Tbms2 > Tbms1.
\end{array}
$$

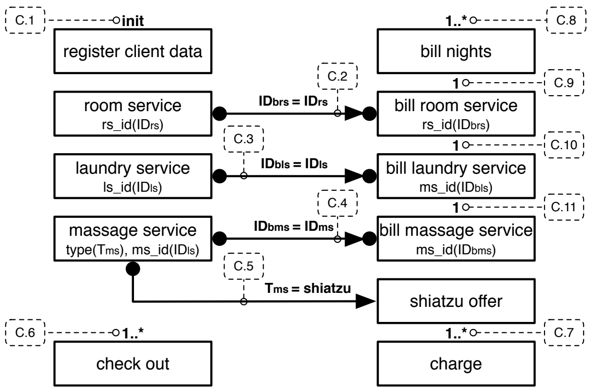


Fig. 2. A ConDec model augmented with a data-related perspective

One thing to observe is that, for constraint ($C.1$), the ConDec init constraint has been mapped in $\mathcal{S}$CIFF by imposing that the *"register_client_data"* activity is expected to happen at time 1 (the first activity in an execution trace).

As described in Section 2.2, $\mathcal{S}$CIFF can be used not only to formalize ConDec, but also to support different extensions to the language. These extensions are useful in the formalization of the hotel and spa process model. Let us consider the following constraints: after having chosen a massage service, this service must be billed to the client and a massage service must be billed only if the client has effectively received the service; if the client has chosen a shiatzu massage, then she can also take advantage of a special offer. In order to link each different service with its specific bill, we attach to the execution of these activities an identifier. Moreover, the second statement deals with a specific execution of the massage service, namely the one in which the client has actually chosen a shiatzu massage; so we attach a type attribute to massage services. This information has been included into the terms representing events in the $\mathcal{S}$CIFF language and the two sentences mapped to the integrity constraints ($C.4$) and ($C.5$).

We could incorporate such a data-related perspective directly at a graphical level, by representing activities together with their data and annotating the ConDec constraints with data condition (such as $TYms = shiatzu$ in $C.5$). Figure 2 shows how the model discussed above can be graphically rendered with annotations.

3 Learning Models

In this section, we describe the approach adopted for mining ConDec models. We first briefly review some concepts of Inductive Logic Programming and the ICL algorithm in particular, then we discuss how ICL has been applied to learning $\mathcal{S}$CIFF constraints and finally we illustrate the DecMiner ProM plug-in for mining $\mathcal{S}$CIFF and ConDec constraints.

3.1 Inductive Logic Programming Techniques

The idea of exploiting Inductive Logic Programming (ILP) for declarative process mining comes form the similarities 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 ILP [15]. Besides the fact that both $\mathcal{S}$CIFF and clausal theories can be used to classify a set of atoms (i.e., an interpretation) as positive or negative, they have strong similarities in the structure of the logical formula composing the theory. Then, thanks to the mapping of ConDec into $\mathcal{S}$CIFF rules, it is possible to learn ConDec models.

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. A set of clauses (i.e. a theory) is true in an interpretation I iff all the clauses are true in I.

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 the least Herbrand model of Clark's completion [5]. By using a background knowledge we are able to encode each interpretation parsimoniously, by storing only once 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 clausal language $\mathcal{L}$, a set P of positive interpretations, a set N of negative interpretations and a definite clause background theory B, we want to find a clausal theory $H \in \mathcal{L}$ such that for all $p \in P$, H is true in the interpretation $M(B \cup p)$, and 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.

The clausal language $\mathcal{L}$ is used in order to restrict the search space. It is usually described in an intensional way using a specific representation language. The description of $\mathcal{L}$ in this language is called *language bias* (LB).

An algorithm that solves the above problem is ICL [6]. In it, a function named Inductive-Constraint-Logic performs a covering loop 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 and the negative examples excluded by it are removed from N. The loop ends when N is empty or when no clause is found.

The clause to be added in every iteration of the covering loop is returned by another procedure (namely, Find-Best-Clause). 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 an initial beam composed of the most specific clauses present in the language bias that is returned by the function MostSpecific(LB). The clauses in the beam are then gradually generalized. The maximum number of generalization steps is a user-defined parameter.

The generality order that is used is θ-subsumption [19], a relationships between two clauses that can be checked syntactically and is stronger than implications. Generalizations of a clause C are obtained by adding a literal to the body or an atom to the head of C. The language bias of ICL defines the literals that can be added to clauses. Moreover, the language bias defines also the set of most specific clauses.

3.2 Application of ICL to $\mathcal{S}$CIFF Learning

ICL has been effectively used to learn $\mathcal{S}$CIFF ICs in Declarative Process Model Learner (DPML) [14]. Each IC is seen as a clause that must be true in all the positive interpretations (compliant execution traces) and false in some negative interpretations (non-compliant execution traces). A theory, composed of a set of 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.

If we define a generality order and a generalization operator for ICs, we can apply an algorithm similar to ICL for learning ICs. The generality order can be defined in this way: 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 $d\theta \subseteq c$.

A generalization of an IC C can be obtained in the following ways: adding a literal to the body, adding a disjunct to the head, removing a literal from a positive disjunct in the head or adding a literal to a negative disjunct in the head. The language bias takes the form of a set of assertions 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.

When adding a disjunct to the head, the generalization 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.

3.3 Learning ConDec Models and $\mathcal{S}$CIFF Rules: The DecMiner Plug-in

DPML has been further extended in this work in order to be able to learn both ConDec models and $\mathcal{S}$CIFF ICs and re engineered as a mining plug-in of the ProM [23] process mining framework, named DecMiner.

DecMiner learns a ConDec model, by first learning $\mathcal{S}$CIFF ICs and then translating them into ConDec constraints using the mapping introduced in Section 2.2. In order to ease the translation, we provide DecMiner with a special language bias that allows only ICs that can be translated into ConDec. We generate this language bias automatically starting from a set of general templates, one for each ConDec constraint, that can be instantiated to generate specific assertions for the language bias. The number of all possible assertions can be huge, while the user could be interested to models defined only by a small, yet meaningful set of ConDec constraints. For this reason, we let the user the possibility of selecting a subset of activities A and a subset of ConDec constraints T. Then, our approach uses only the instantiation of these constraints with the selected activities for learning the model. Besides providing as output a model that fits the user requirements, smaller constraint sets allow also better performances of the learning algorithm.

The accuracy and learning time depends on the choice of these subsets. They influence the accuracy of the learned model because an activity relation discriminating between compliant and non-compliant execution traces cannot be learned if the appropriate template and/or activities were not chosen. The time complexity is linear in the number of traces and in the number of constraints. With respect to the number of activities, it is quadratic if there are binary constraints, and linear if there are only unary constraints.

An advantage of mining ConDec constraints through $\mathcal{S}$CIFF is that the approach can be extended to induce constraints involving more than two activities, for example constraints having a conjunction of preconditions or a disjunction of postconditions, and constraints with conditions over data.

DecMiner implements all the data preparation and learning phases of the mining process described above and guides the user by means of its graphical user interface. In the first phase, named “Classification”, the user uses the graphical interface shown in Figure 3 to browse the execution traces and label some of them as compliant (positive) or not compliant (negative). In the second phase, named “Activities”, the user can choose among all the activities and their associated parameters the information that she considers important for learning the declarative model. In the third phase, named “Templates”, the user uses the graphical interface shown in Figure 4 to choose the set of existence, relation and negation ConDec templates to be used in the mining phase. The fourth phase, named “Mining”, is started when the “Start mining” button is pressed. In this phase the language bias for ICL is generated, by instantiating the chosen templates with the chosen activities, and the learning algorithm is applied, producing the declarative model. In the fifth phase, named “Results”, the learned

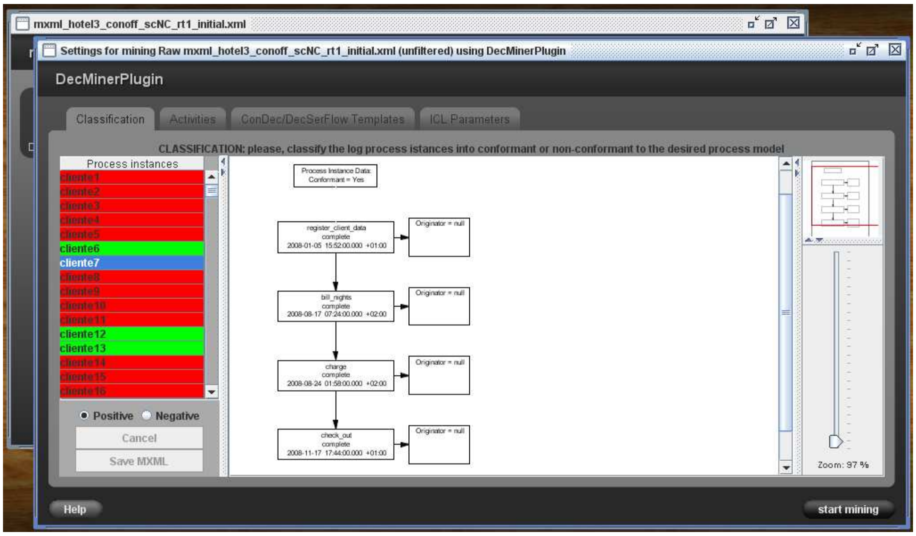

mxml_hotel3_conoff_scNC_rt1_initial.xml
Settings for mining Raw mxml_hotel3_conoff_scNC_rt1_initial.xml (unfiltered) using DecMinerPlugin
DecMinerPlugin
Classification
Activities
ConDec/DecSerFlow Templates
ICL Parameters
CLASSIFICATION: please, classify the log process istances into conformant or non-conformant to the desired process model
Process instances
cliente1
cliente2
cliente3
cliente4
cliente5
cliente6
cliente7
cliente8
cliente9
cliente10
cliente11
cliente12
cliente13
cliente14
cliente15
cliente16
Positive
Negative
Cancel
Save MXML
Process Instance Data: Conformant = Yes
register_client_data complete 2008-01-05 15:52:00.000 +01:00
bill_nights complete 2008-08-17 07:24:00.000 +02:00
charge complete 2008-08-24 01:58:00.000 +02:00
check_out complete 2008-11-17 17:44:00.000 +01:00
Originator = null
Originator = null
Originator = null
Originator = null
Zoom: 97 %
Help
start mining

Fig. 3. DecMiner plug-in: trace classification

$\mathcal{S}$CIFF rules and ConDec constraints are presented to the user[1]. The current version of the tool can be downloaded from the web[2].

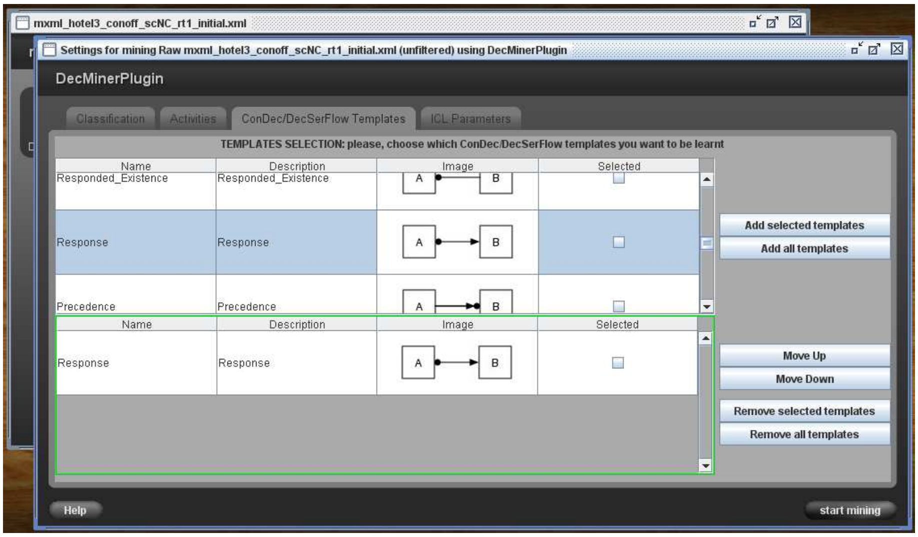
4 Experiments

In order to evaluate the effectiveness and robustness of the learning approach, we followed a typical machine learning experimentation methodology: first, we create an artificial process model, described in Section 2.3, which presents some difficulties for the learning approach to be tested; second, we randomly generate from such model several training and testing datasets; third, we apply the learning approach on the training datasets obtaining models; finally, we compare the learned models with the original one and compute the classification accuracy of the learned models on the testing datasets.

Given the ConDec and $\mathcal{S}$CIFF process models described in Section 2.3, we generated five training and five testing datasets. The generation of each of them is made in two phases. In the first, a Java application randomly creates an execution trace. In the second phase, the $\mathcal{S}$CIFF Checker (presented in Section 2) is used to classify each trace as compliant or non-compliant with respect to the correct hotel process. The process is repeated until a dataset containing 2000 compliant traces and 2000 non-compliant traces has been generated.

[1] The ConDec model is shown by using the DECLARE tool `http://is.tm.tue.nl/research/declare/`

[2] `http://www.unife.it/dipartimento/ingegneria/informazione/informatica/processmining/`



mxml_hotel3_conoff_scNC_rt1_initial.xml

Settings for mining Raw mxml_hotel3_conoff_scNC_rt1_initial.xml (unfiltered) using DecMinerPlugin

TEMPLATES SELECTION: please, choose which ConDec/DecSerFlow templates you want to be learnt

Name	Description	Image	Selected
Responded_Existence	Responded_Existence	A —● B	☐
Response	Response	A ●→ B	☐
Precedence	Precedence	A →● B	☐

Name	Description	Image	Selected
Response	Response	A ●→ B	☐

Fig. 4. DecMiner plug-in: ConDec template selection

DecMiner has been then applied to each training dataset. By applying the learned models to the classification of the testing sets, we computed the *classification accuracy*, defined 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. The average accuracy achieved by DecMiner was 100%.

Comparing the original hotel process model with those learned by using the five training sets, we observe that sometimes there are differences in the learned constraints. This happens because some of the randomly generated training sets do not contain the traces that allow to distinguish the behaviors of similar constraints.

This is the case, for instance, of the precedence(A,B) and responded_existence(A,B) constraints. They share a common set of labeled execution traces (BA labeled as compliant, B labeled as compliant and A labeled as non-compliant) and cannot be distinguished until a trace containing AB is labeled as compliant or non-compliant. In the first case, DecMiner learns a responded_existence(A,B) constraint otherwise it learns a precedence(A,B). In real applications, this behavior can be considered an advantage because it allows dynamic adaptation and refinement of the learned process models when new traces are classified as compliant and non-compliant and added to the training set. If the traces distinguishing the behavior of two constraints are not present in the dataset, DecMiner learns the constraint that comes first in the language bias.

We also investigated the robustness of DecMiner to noise in the classification of traces: we repeated the experiments by considering training sets with an increasing portion of misclassified examples. Table 2 shows that the performances of DecMiner degrade gracefully with the increase of the amount of noise.

Table 2. Accuracy as a function of noise

Errors	Accuracy
2%	99.72 %
10%	98.85 %
20%	97.29 %

As for other ILP systems, the learning phase depends not only on the training set but also on the language bias: by restricting it to different subsets of the ConDec constraints, it is possible to learn different models. Each model corresponds to a different perspective about a real process, pointing out different aspects. The 100% accuracy achieved in the former experiment is a consequence of the chosen language bias: all the templates were added with those referring to the constraints in the original hotel and spa model put at the top of the language bias. We evaluated the influence of language bias on classification accuracy, by randomly mixing the ConDec templates in the bias. Despite this change, the accuracy achieved by DecMiner considering datasets without noise remains high (99.97%).

Results achieved by our approach on other real and artificial datasets (e.g., the cervical cancer screening process, the netbill e-commerce protocol and an auction protocol) are reported in [14] and [13].

5 Related Works and Discussion

Process mining is an active research field. Notable works in such a field are [1,21,24,11,7,9]. Agrawal et al. [1] introduced the idea of applying process mining to workflow management. The authors proposed an approach for inducing a process representation in the form of a directed graph encoding the precedence relationships. van der Aalst et al. [21] presented the α-algorithm for inducing Petri nets from data and identified 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 [24] van Dongen and van der Aalst described an algorithm which derives causal dependencies between activities and uses them for constructing instance graphs, presented in terms of Event-driven Process Chains. [11] is a recent work where a process model is induced in the form of a disjunction of special graphs called workflow schemas.

We differ from these works because we use a representation that is declarative rather than procedural, without sacrificing expressiveness. Moreover, we learn from compliant and non-compliant traces, rather than from compliant traces.

[7,9] are closer to our work because they deal with mining (partially) declarative specifications. In [7] the learning starts from process *runs* that are high level specification of a set of process traces and are represented by means of Petri nets. Mining is performed by merging the different runs for the same process. The model that is obtained is hybrid, in the sense that it may contain sets of activities that must be executed but for which no specific order is required. We

differ from this work because we start from traces rather than runs: while runs specify already a partial order among activities, traces are simply a sequence of events representing activity executions. Therefore, runs are already very informative of the process model.

[9] related BPM to the field of planning in artificial intelligence: activities in business process are seen as planning operators with pre-conditions and post-conditions. Representing a process in this way requires the specification of *fluents* besides activities, i.e., properties of the world that may change their truth value during the execution of the process. The adoption of fluents allows to explicitly express pre-conditions and post-conditions of activities. Thus fluents introduce a new dimension to BPM that needs further explorations. Our work remains in the traditional domain of BPM in which the pre-conditions and post-conditions of activities are left implicit. The approach for learning process models of [9] involves iterating planning and operator refinement: given the current definition of the pre-conditions and post-conditions of the activities, 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. In order to avoid asking the user to classify activities, [10] proposed an approach for automatically generating negative events, i.e., events that are used as negative examples. In the future we plan to investigate the extension of this approach to the automatic generation of negative traces.

With respect to performance evaluation, a direct comparison with [21,24] is unfair since we adopted accuracy (since we have compliant and non-compliant traces in the test set) while they adopt fitness.

In this special issue, [22] and [8] face the problem of mining a process representation in the form of a Petri net, while [20] extracts metrics and patterns from collaborative processes in SOA-based environments.

6 Conclusions and Future Work

We propose a methodology for analyzing a log containing several traces labeled as compliant or non-compliant. From them we learn a set of declarative constraints expressed as $\mathcal{S}$CIFF rules able to accurately classify a new trace, and corresponding to a ConDec model.

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 can learn a model which expresses also what is forbidden. Furthermore, the learned $\mathcal{S}$CIFF ICs are easily mapped into ConDec constraints. We call the resulting system DecMiner.

In order to test the proposed methodology, we performed an experiment on a case study regarding the management of a hotel and spa. The results show that DecMiner nearly recovers the correct model. Other experiments have been documented in [14,13].

In the future, we plan to apply DecMiner to university students' careers, where positive traces are careers of students that graduated on time, and negative ones are careers of students who did not finish their studies in the prescribed time.

Moreover, we plan to investigate the development of a *mining-checking cycle*, in which learning is interleaved with classification of traces into positive or negative either manually by the user or automatically using the $\mathcal{S}$CIFF Checker plug-in with a user specified model. In this way the user can improve an initial model of the process by experimenting different languages biases.

Acknowledgments. This work has been partially supported by the FIRB project "TOCAI.IT" and FAR projects by University of Bologna and University of Ferrara. We would like to thank Pamela Zapparoli for her contribution to DecMiner development.

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., Torroni, P.: Verifiable agent interaction in abductive logic programming: the SCIFF framework. ACM T. Comput. Logic 9(4) (2008)
3. Chesani, F., Mello, P., Montali, M., Riguzzi, F., Sebastianis, M., Storari, S.: Checking compliance of execution traces to business rules. In: Ardagna, D., et al. (eds.) BPM 2008 Workshops. LNBIP, vol. 17, pp. 134–145. Springer, Heidelberg (2009)
4. 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)
5. Clark, K.L.: Negation as failure. In: Logic and Databases. Plenum Press (1978)
6. De Raedt, L., Van Laer, W.: Inductive constraint logic. In: Zeugmann, T., Shinohara, T., Jantke, K.P. (eds.) ALT 1995. LNCS (LNAI), vol. 997, pp. 80–94. Springer, Heidelberg (1995)
7. Desel, J., Erwin, T.: Hybrid specifications: looking at workflows from a run-time perspective. Int. J. Computer System Science & Engineering 15(5), 291–302 (2000)
8. Bergenthum, R., Desel, J., Mauser, S., Lorenz, R.: Construction of process models from example runs. In: Jensen, K., van der Aalst, W. (eds.) ToPNoC II. LNCS, vol. 5460, pp. 243–259. Springer, Heidelberg (2009)
9. 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), 485–505 (2006)
10. Goedertier, S.: Declarative techniques for modeling and mining business processes. PhD thesis, Katholieke Universiteit Leuven, Faculteit Economie en Bedrijfswetenschappen (2008)
11. Greco, G., Guzzo, A., Pontieri, L., Saccà, D.: Discovering expressive process models by clustering log traces. IEEE Trans. Knowl. Data Eng. 18(8), 1010–1027 (2006)
12. Jaffar, J., Maher, M.J.: Constraint logic programming: a survey. J. Logic Program. 19(20), 503–582 (1994)

13. Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Inducing declarative logic-based models from labeled traces. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 344–359. Springer, Heidelberg (2007)
14. Lamma, E., Mello, P., Riguzzi, F., Storari, S.: Applying inductive logic programming to process mining. In: Blockeel, H., Ramon, J., Shavlik, J., Tadepalli, P. (eds.) ILP 2007. LNCS, vol. 4894, pp. 132–146. Springer, Heidelberg (2008)
15. Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. J. Logic Program. 19(20), 629–679 (1994)
16. Pesic, M.: Constraint-Based Workflow Management Systems. PhD thesis, Technische Universiteit Eindhoven, Department of Technology Management (2008)
17. 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, pp. 287–300. IEEE Computer Society, Los Alamitos (2007)
18. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)
19. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM 12(1), 23–41 (1965)
20. Truong, H.L., Dustdar, S.: Online interaction analysis framework for ad-hoc collaborative processes in SOA-based environments. In: Jensen, K., van der Aalst, W. (eds.) ToPNoC II. LNCS, vol. 5460, pp. 260–277. Springer, Heidelberg (2009)
21. 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)
22. van Dongen, B., de Medeiros, A.K.A., Wen, L.: Process mining: Overview and Outlook of Petri net discovery algorithms. In: Jensen, K., van der Aalst, W. (eds.) ToPNoC II. LNCS, vol. 5460, pp. 225–242. Springer, Heidelberg (2009)
23. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The ProM framework: A new era in process mining tool support. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 444–454. Springer, Heidelberg (2005)
24. 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)