
Learning DecSerFlow Models from Labeled Traces

Evelina Lamma
Fabrizio Riguzzi
Sergio Storari

ENDIF, Università di Ferrara, Via Saragat, 1, 44100 Ferrara, Italy

Paola Mello
Marco Montali

DEIS, Università di Bologna, Viale Risorgimento, 2, 40136 Bologna, Italy

EVELINA.LAMMA@UNIFE.IT
FABRIZIO.RIGUZZI@UNIFE.IT
SERGIO.STORARI@UNIFE.IT

PMELLO@DEIS.UNIBO.IT
MMONTALI@DEIS.UNIBO.IT

Abstract

We present the system DecMiner that induces DecSerFlow models from positive and negative traces. The approach we follow consists in first inducing *SCIFF* constraints and then converting them into DecSerFlow ones.

1. Introduction

DecSerFlow (van der Aalst & Pesic, 2006) is a recent language for expressing process models in a declarative way: DecSerFlow captures what is the high-level process behaviour without expressing how it is procedurally executed, hence giving a concise and easily interpretable feedback to the business manager.

In this paper, we propose an approach for mining DecSerFlow models starting from a set of execution traces, previously labeled as compliant or not to the process. The approach consists in first inducing logical formulas in the *SCIFF* language (Alberti et al., 2007) and then converting these formulas into DecSerFlow.

SCIFF is 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. *SCIFF* models interaction patterns with integrity constraints that state what is expected to be performed or what is forbidden when a given condition, expressed in terms of already performed activities, holds. *SCIFF* theories can be used to classify a trace as compliant or not with the model. The trace is represented by an interpretation.

Appearing in *Proceedings of the 24th International Conference on Machine Learning*, Corvallis, OR, 2007. Copyright 2007 by the author(s)/owner(s).

An important advantage of adopting a computational logic representation is that it is possible to exploit the techniques developed in the field of Inductive Logic Programming for learning models from examples and background knowledge. In this work, we adapt the system ICL (De Raedt & Van Laer, 1995) to the problem of learning a simplified version of *SCIFF* constraints. In fact, a *SCIFF* theory can be seen as a set of clauses each of which must be true in a trace for it to be classified as compliant. In order to apply ICL, we defined: a simple procedure for testing the truth of a *SCIFF* clause, a generality relation between constraints and a refinement operator. We called the resulting system DecMiner (Declarative Miner).

The conversion from *SCIFF* to DecSerFlow is performed by inverting the translation from DecSerFlow to *SCIFF* proposed in (Chesani et al., 2007). By following this approach we do not mine a complete process model, but rather discover a set of common declarative patterns and constraints.

The language bias we consider in this work is defined in terms of a set of templates, each of which specifies which literals can be added or removed from *SCIFF* formulas. Each template refers to a DecSerFlow constraint. When the learning terminates, the integrity constraints can be automatically translated into DecSerFlow constraints.

We demonstrate the viability of the approach by applying it to a cervical cancer screening process.

2. An Overview of the *SCIFF* Framework

The *SCIFF* framework (Alberti et al., 2007) was originally developed for the specification and verification of agent interaction protocols within open and hetero-

geneous 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. SCIFF 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 , we use the atom $ev(T)$ where T is an integer. If there are arguments for the event, they are included in the atom. Hence, an execution trace is modeled as a set of atoms. For example, we could formalize that *bob* has performed activity a at time 5 as follows: $a(bob, 5)$.

In this paper, we consider a syntax of ICs that is a subset of the one in (Alberti et al., 2007). In this simplified syntax, a Social Integrity Constraint, C , is a logical formula of the form

$$\begin{aligned} \text{Body} \rightarrow \exists(\text{Conj}P_1) \vee \dots \vee \exists(\text{Conj}P_n) \vee \\ \forall \neg(\text{Conj}N_1) \vee \dots \vee \forall \neg(\text{Conj}N_m) \end{aligned} \quad (1)$$

where Body , $\text{Conj}P_i$ $i = 1, \dots, n$ and $\text{Conj}N_j$ $j = 1, \dots, m$ are conjunctions of literals built over event atoms, over predicates defined in the background or over built-in predicates. The quantifiers in the head apply to all the variables not appearing in the body. The variables of the body are implicitly universally quantified with scope the entire formula.

We will use $\text{Body}(C)$ to indicate Body and $\text{Head}(C)$ to indicate the formula $\exists(\text{Conj}P_1) \vee \dots \vee \exists(\text{Conj}P_n) \vee \forall \neg(\text{Conj}N_1) \vee \dots \vee \forall \neg(\text{Conj}N_m)$ and call them respectively the *body* and the *head* of C . We will call P *conjunction* each $\text{Conj}P_i$ for $i = 1, \dots, n$ and N *conjunction* each $\text{Conj}N_j$ for $j = 1, \dots, m$. We will call P *disjunct* each $\exists(\text{Conj}P_i)$ for $i = 1, \dots, n$ and N *disjunct* each $\forall \neg(\text{Conj}N_j)$ for $j = 1, \dots, m$.

An example of an IC is

$$\begin{aligned} a(bob, T) \wedge T < 10 \\ \rightarrow \exists T_1 (b(alice, T_1) \wedge T < T_1) \\ \vee \\ \forall T_1 \neg(c(mary, T_1) \wedge T < T_1 \wedge T_1 < T + 10) \end{aligned} \quad (2)$$

The interpretation of an IC is the following: if there exists a substitutions of variables such that the body is true in an interpretation representing a trace, then at least one of the disjuncts in the head must be true.

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 a time later than T or we expect that *mary* does not execute action c within 9 time units after T .

3. A Brief Description of DecSerFlow

In this section we will briefly introduce the Declarative Service Flow language (DecSerFlow). For a detailed description of the language and its mapping to Linear Temporal Logic, see (van der Aalst & Pesic, 2006).

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.

The basic intuitive concepts of DecSerFlow are: activities (atomic units of work) and 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 “formulae” to indicate DecSerFlow relationships.

DecSerFlow core relationships are grouped into three families: *existence formulae*, unary relationships used to constrain the cardinality of activities; *relation formulae*, which define (positive) relationships and dependencies between two (or more) activities, and *negation formulae*, the negated version of relation formulae (as in SCIFF, 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 formulae can be expressed by using SCIFF. In (Chesani et al., 2007), the authors propose a preliminary translation by mapping atomic DecSerFlow activities to SCIFF events and formulae to corresponding integrity constraints.

Each DecSerFlow pattern is translated into one or more ICs. For example, let us consider the response relation $\text{response}(a, b)$, shown in Figure 1(a), whose meaning is: every time a is performed, b should be performed *after* it. This relation is formalized into the following Integrity Constraint:

$$a(T_a) \rightarrow \exists T_b (b(T_b) \wedge T_b > T_a). \quad (3)$$

Another example is the succession relation $\text{succession}(a, b)$, shown in Figure 1(b), whose meaning is: every execution of a should be *followed* by the execution of b and each b should be *preceded* by a . This relation is formalized into two Integrity

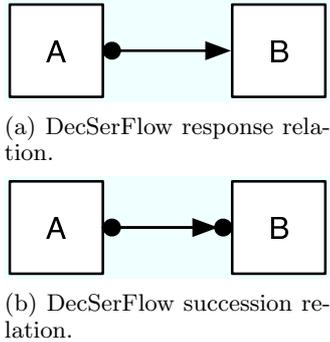


Figure 1. Two different DecSerFlow service relations.

Constraints, IC (3) and:

$$b(T_b) \rightarrow \exists T_a (a(T_a) \wedge T_a < T_b). \quad (4)$$

4. Learning DecSerFlow Models

This work starts from the idea that there is a similarity between learning a SCIFF 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. In fact, as a SCIFF 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.

An algorithm that solves the learning from interpretation problem is ICL (De Raedt & Van Laer, 1995). It performs a covering loop in which negative interpretations are progressively ruled out and removed from the training set. 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 there are no more negative examples or when no clause is found.

The clause to be added in every iteration of the covering loop is found 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 generality order that is used is the θ -subsumption order. The literals that can possibly be added to a clause for refining it are specified in the *language bias*,

a collection of statements in an ad hoc language that prescribe which refinements have to be considered.

In (Lamma et al., 2007b) we have proposed an approach for applying ICL to the problem of learning ICs. Each IC is seen as a clause that must be true on all the positive traces and false on some negative ones. The theory composed of all the ICs must be such that all the ICs are true when considering a positive trace and at least one IC is false when considering a negative 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 conjunction d in the head of D : if d is positive, then there exist a positive conjunction c in the head of C such that $d\theta \supseteq c$, if d is negative, then there exist a negative conjunction c in the head of C such that $d\theta \subseteq c$.

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 or adding a literal to a negative disjunct in the head.

We have chosen to provide the language bias in 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. A preliminary version of this approach appears in (Lamma et al., 2007a). We call the system implementing this approach DecMiner.

5. Experiments

As a case study for exploiting the potentialities of our approach we choose the process of cervical cancer screening (CERV, 2007) 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; Sec-

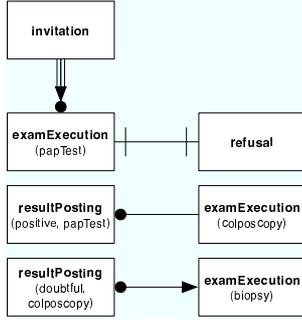


Figure 2. Mined DecSerFlow representation.

Table 1. Results of the experiments.

Experiment	DecMiner	α algorithm
Screening	97.44 %	96.15 %

ond level test with colposcopy, and eventually biopsy.

For the screening process, we have a total of 55 positive traces and 102 negative traces. Five fold cross validation was performed.

The results are compared with those of the α algorithm (van der Aalst & van Dongen, 2002). The average accuracy of the two approaches is shown in Table 1. Since the α algorithm learns from positive traces only, the accuracy was measured by learning from positive traces only and then applying the mined model to both the positive and the negative test traces.

Moreover, we have applied DecMiner to the whole dataset and we have manually translated the mined patterns to DecSerFlow. The resulting model is shown in Figure 2.

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 (Chesani et al., 2007).

6. Acknowledgements

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

References

Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., & Torroni, P. (2007). Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Trans. on Computational*

Logics. Accepted for publication.

CERV (2007). Cervical cancer screening web site. Available at: <http://www.cancer.gov/cancertopics/pdq/screening/cervical/healthprofessional>.

Chesani, F., Mello, P., Montali, M., & Storari, S. (2007). *Towards a DecSerFlow declarative semantics based on computational logic* (Technical Report DEIS-LIA-07-002). DEIS.

De Raedt, L., & Van Laer, W. (1995). Inductive constraint logic. *Proc. of the 6th Conf. on Algorithmic Learning Theory*. Springer Verlag.

Lamma, E., Mello, P., Montali, M., Riguzzi, F., & Storari, S. (2007a). Inducing declarative logic-based models from labeled traces. *Proceedings of the 5th International Conference on Business Process Management*. Springer.

Lamma, E., Mello, P., Riguzzi, F., & Storari, S. (2007b). Applying inductive logic programming to process mining. *Proceedings of the 17th International Conference on Inductive Logic Programming*. Springer.

van der Aalst, W. M. P., & Pesic, M. (2006). DecSerFlow: Towards a truly declarative service flow language. *Proc. of the 3rd Int. Workshop on Web Services and Formal Methods*. Springer.

van der Aalst, W. M. P., & van Dongen, B. F. (2002). Discovering workflow performance models from timed logs. *First International Conference on Engineering and Deployment of Cooperative Information Systems* (pp. 45–63).