Revising Process Models through Inductive Learning

Fabrizio Maria Maggi[1], Domenico Corapi[2] Alessandra Russo[2], Emil Lupu[2], and Giuseppe Visaggio[3]

[1] Department of Mathematics and Computer Science, University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
f.m.maggi@tue.nl

[2] Department of Computing, Imperial College London, 180 Queen's Gate, SW7 2AZ London, UK
{d.corapi,a.russo,e.c.lupu}@imperial.ac.uk

[3] Department of Computer Science, University of Bari, Via Orabona 4, 70126 Bari, Italy
visaggio@di.uniba.it

Abstract. Discovering the Business Process (BP) model underpinning existing practices through analysis of event logs, allows users to understand, analyse and modify the process. But, to be useful, the BP model must be kept in line with practice throughout its lifetime, as changes occur to the business objectives, technologies and quality programs. Current techniques require users to manually revise the BP to account for discrepancies between the practice and the model, which is a laborious, costly and error prone task. We propose an automated approach for resolving such discrepancies by minimally revising a BP model to bring it in line with the activities corresponding to its executions, based on a non-monotonic inductive learning system. We discuss our implementation of this approach and demonstrate its application to a case-study. We further contrast our approach with existing BP discovery techniques to show that *BP revision* offers significant advantages over *BP discovery* in practical use.

Key words: Information systems, processes, inductive learning, maintenance of process models.

1 Introduction

In numerous applications a Business Process (BP) model must be uncovered from existing procedures and practices. The effort required to acquire and adapt models has been estimated to amount to around 60% of the total development time [6]. Thus, a variety of techniques have been proposed for mining process models from event logs of executed activities as recorded by information systems [12]. Event logs typically contain rich information about events occurred

during the process execution. Process mining approaches have shown that this information can be used to construct models of the underlying BP (i.e. *process discovery*) [13].

However, once uncovered, the model must remain a faithful representation of the reality even in the face of changes to the underlying procedures and practices. This requires users either to *re-discover* the process or to identify *discrepancies* and *revise* the BP model to address them. The first option may not be optimal in real applications because the techniques employed so far may re-discover models that differ significantly from those previously learnt, and any analysis performed on those models needs to be redone entirely. The second option, has led to *conformance testing* approaches that can identify and evaluate the discrepancies between existing models and actual process executions [9]. But when discrepancies are detected, the analyst has to manually apply changes to the process model to reconcile the model with the actual execution. This can be a difficult, costly and error prone task that relies mainly on the effort and expertise of the analyst. The task is even harder for models where different processes must cooperate within the frame of a set of constraints [11].

In this paper, we propose an approach for automated *revision* of process models. It takes as input a set of event logs corresponding to the actual execution of the tasks (and thus considered positive examples of the actual process), and an existing process model (either specified by an expert or learnt in a previous iteration). Our approach then minimally revises the existing process model to account for the discrepancies between the model and the logs. This has the advantage of giving users a model which is "close" to the one they have previously used, thus enabling them to re-use the analysis and reasoning previously conducted, whilst highlighting the changes necessary to account for the new log entries. Our work builds on AGNEs (Artificially Generated Negative Events) [5], a logic based approach for discovering business process, from which we reuse the logical formalisation of the process and the method for generating the training data. The latter includes generating from the logs negative examples that account for executions *not* present in the logs. Our main contribution in this paper consists in a novel framework for the revision of process models based on non-monotonic inductive logic programming (NMILP). Whenever discrepancies are detected between the current model and the logs, an inductive learning system is used to compute *changes* and automatically suggest revised models that would resolve the detected inconsistencies. The implementation of the learning system guarantees *minimal changes* to the existing model, and in particular that all aspects of the model unaffected by the discrepancies will be preserved in the revised model. This would not necessarily be the case if the model was simply re-discovered from the event logs.

We present and employ in an exemplifying case study a new NMILP system called *TAL*[3], that compared to AGNEs (where a hill-climbing search is performed) introduces an explicit semantics for negation and a different search method, based on a thorough exploration of the space of the solution. In essence, our solution trades efficiency for soundness and in turn enables effective learning

with less training data. With respect to traditional process discovery techniques we inherit all the advantages of AGNEs such as a richer representation language that enables learning more complex process models (e. g. time-varying properties and history-dependent conditions).

This paper is organized as follows. Section 2 describes the main features of our proposed approach. Section 3 details the revision algorithm. Section 4 provides an illustrative case study on a real application domain. A summary and some remarks about future work conclude the paper.

2 Approach

Starting from an event log and an existing BP model (encoded as a Petri net) that is not in line with the log, our approach provides a systematic and automated way for learning minimal revisions to the model so that the revised Petri net fits the logs. More formally, a process revision task can be defined as follows:

Definition 1 (Process model revision task).
Given an event log W of execution instances of a BP and an existing Petri net P modelling the BP, the process model revision is the task of finding a Petri net $P^{'}$ that minimally revises *P according to W.*

What is considered "minimal" and the metrics that define the level of conformance of P' wrt W characterise the task and must be defined. It is beyond the scope of this paper to discuss the concept of *minimal revision of a Petri net*. We assume it to be expressed in terms of *minimal revision of a logic program* equivalent to the Petri net, as explained in more detail in Section 3. Informally, adding or deleting a minimum number of conditions and adding or deleting a minimum number of rules in the logical translation of a Petri net, results in revisions that do not add or delete activities or relationships between activities if it is not strictly required.

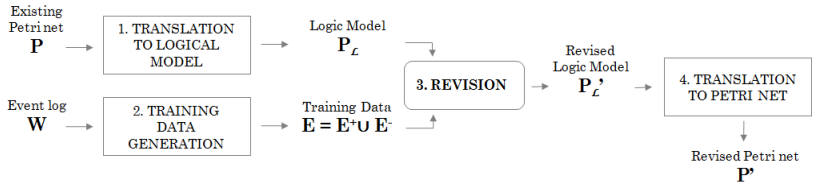

Fig. 1: Approach

Our approach is composed of four phases (see Fig. 1). In the first step the existing Petri net P is automatically translated into a logic program $P_{\mathcal{L}}$. n the second phase a training data set $E = E^{+} \cup E^{-}$ (with positive and negative examples) is generated from the event log W. Whilst positive examples are

naturally derived from the log, the negative examples are artificially generated. $P_{\mathcal{L}}$ and E are then used in the third phase by an Inductive Logic Programming (ILP) system to compute the revision task. The output of this revision task is a logic program $P'_{\mathcal{L}}$ that minimally revises $P_{\mathcal{L}}$. This is then translated, in the fourth phase, back into a Petri net, P', that represents the revised BP model.

The formalisation of Petri nets into logic programs and the generation of training data follow the approach described in [5]. Though we provide, for completeness of our presentation, a brief description of these two steps we refer the reader to [5] for further details.

Before presenting the individual phases of the approach in detail, we briefly summarise the notations and terminology used throughout the paper.

2.1 Notation and Terminology

Given a logic-based alphabet consisting of variables, constants and predicates, an *atom* is an expression of the form $p(t_1, .., t_n)$, where p is a predicate and t_i are terms (variable or constants) in the alphabet. A *negated atom* is an expression of the form $\neg p(t_1, .., t_n)$, where $\neg$ (or equivalently "not") is the Prolog negation-as-failure (NAF) operator [1] and $p(t_1, .., t_n)$ is an atom. A *literal* is either an atom or a negated atom; we will refer to it as *positive* and *negative* literal respectively. A set $\{l_1, ..., l_m\}$ of literals is a *clause*, which is also denoted, in logic programming, as the rule

$$h \leftarrow b_1, ..., b_n$$

where h is positive literal, called the *head* of the clause, and $b_1, ..., b_n$ is a conjunction of literals, called *body* of the clause. Each b_i is also referred to as *condition* or *antecedent* of the rule. The intuitive meaning of a clause is "if all the conditions are true then the head must be true". Using Prolog convention [10], predicates, terms and functions are denoted with initial lower case letter, whereas variables are written with an initial capital letter. Clauses can be of two types, *definite* and *normal*. The former are clauses whose body literals are all positive, the latter as clauses whose conditions can be either positive or negative literals. Clauses with a single literal (the head) are called *facts*, whereas clauses with a body and an empty head are called *goals*. A *normal logic program* is thus a finite set of normal clauses $\{c_1, ..., c_n\}$ assumed to be in conjunction with each other. In the remainder of the paper the symbol $\models$ denotes the notion of entailment over stable model semantics for normal logic programs [4] (equivalent to logical entailment for definite programs).

2.2 Logical translation of a Petri Net

The first phase of our approach translates automatically a Petri net into a normal logic program. The formalisation is based on a predicate ns ("no-sequel") defined as follows:

$$ns(AT1, AT2, BId, Now) \leftarrow \\ event(AT1, BId, completed, AgentId, Parameters, T1), T1 < Now, \\ \neg eventFromTill(AT2, BId, completed, T1, Now)) \quad (1)$$

$$eventFromTill(AT, BId, ET, From, Till) \leftarrow \\ event(AT, BId, ET, AgentId, Parameters, T), From < T, T < Till \quad (2)$$

where the predicate *event* is defined through the logical formalization of a state transition (as explained in the next section)

Using the *ns* predicate any Petri net can be translated into a normal logic program, by expressing each Petri net *transition* in terms of the preconditions under which the transition can take place [5].

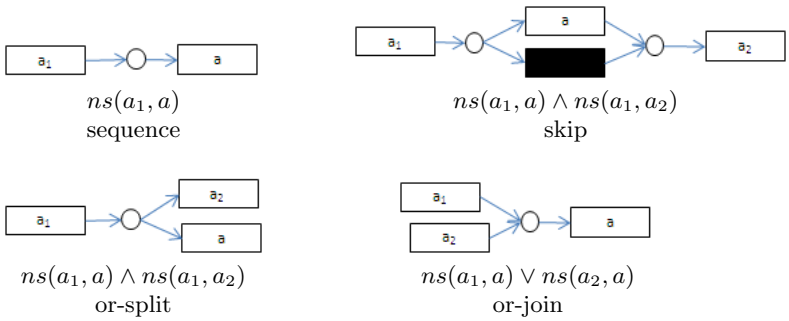

Fig. 2: Formalization of Petri net constructs

In the rest of the paper, we consider only the completion of activities (i.e. events of type *completed*) and we abbreviate the predicate $ns(AT1, AT2, BId, Now)$ to $ns(AT1, AT2)$, whenever there is no ambiguity about the process instance id and the time.

Figure 2 shows the patterns used to map basic constructs of a Petri net in terms of *ns* predicate for the activity a. The rules defining the preconditions for a certain activity a in the logic program $P_{\mathcal{L}}$ are of the type

$$class(a, BId, T, completed) \leftarrow ns(...), ..., ns(...)$$

2.3 Training data generation

In the encoding of an event log into logic programs each state transition is represented using the predicate $event(AT, BId, ET, AgentId, Par, TS)$ where AT represents the activity name, BId is the unique id of the corresponding process instance, ET denotes the event type, $AgentId$ the agent who has performed the state transition, Par a lists of additional parameters and TS is the time

point at which the state transition has happened. In the rest of the paper only the activity name, the process instance id and the time point are used in the revision. However, other arguments can be used to revise richer models than the ones considered here.

To allow the process models revision through supervised multi-relational learning, negative information is also required. A *negative example* defines state transitions that cannot take place. Our training data set generation uses the algorithm proposed in [5] for extracting negative information from given event logs. Briefly, given a process instance t_i in an event log and a state transition $e_{(i,k)}$ in t_i, the algorithm checks the occurrence of any other state transition, ϵ, in the position k. If there exists a process instance t_j: $\forall l, l < k, similar(e_{(i,l)}, e_{(j,l)})$ and $similar(e_{(j,k)}, \epsilon)$ then the state transition ϵ is not added as negative information (because this behaviour is present in the event log). If such transition t_j doesn't exist, then ϵ is added as a negative state transition at position k.

3 Revision

The revision phase takes as input the logic program representation of an existing Petri net and the training set data, and generates as output a new logic program that covers all the positive examples and none of the negative examples (i.e. $TP = 1$ and $TN = 1$). The algorithm uses an underlying non-monotonic ILP system to find, as inductive solutions, prescriptive syntactic changes to be made to the original model. The computation of such changes is performed within a search space defined by a *language bias*, given as input to the underlying learning system, which defines the syntactic form of the possible changes that can be learned. In contrast to the hill climbing learning approaches used in AGNEs, our learning system explores the entire search space, and therefore it always finds a solution, if one exists.

Let us now define our notion of revision through learning.

Definition 2 (Revision through learning).

Given a revisable set T of rules, a background knowledge B (not revisable), a set E of positive and negative examples and a language bias L, revision through learning *is the task of finding a set of* minimal changes*, within the scope of L, that when applied to T gives a revised set T' of rules that, together with the background knowledge B, covers all the positive examples and none of the negative one.*

The key notion in the above definition is that of *minimal change*. In general, a revision system avoids the computation of new models that are "unrelated" to the revisable part of the original model. Therefore, whenever an initial (even if not correct) model exists, either because provided by an expert or available from previous revisions, minimal revision is, in general, preferable to rediscovery. Our revision algorithm uses a measure of minimality similar to that proposed in [14], and defined in terms of *number of revision operations* required to transform one model into another.

In our approach, computing a *minimally revised* Petri net $P^{'}$ from a given Petri net P, corresponds to computing a logic program $T^{'}$ that can be obtained from the logic program T representing P, by means of a minimal number of atomic revision operations.

3.1 Revision algorithm

Our revision algorithm takes as input a logic program T representation of a Petri net model (as revisable model), a background knowledge B, a set E of examples and a language bias L. It then produces as output a revised logic program $T^{'}$ using three main computation steps ([2]). At first, during the *pre-processing* phase, all the rules in the given revisable program are transformed into defeasible rules. This step intuitively changes the meaning of the rules from "the head of a rule is true if all the conditions are true" to "the head of a rule is true if all the conditions are true and the exception to the rule is not true". Defining an exception for a revised rule is equivalent to add conditions to it.

The second step of the algorithm is the *learning* phase. This takes the transformed revisable program generated by the pre-processing phase and computes the revision in terms of conditions that can be added and or deleted from the transformed rules to cover the given positive examples and non of the negative one. This phase uses a prototype non-monotonic learning systems called TAL (*Top-directed Abductive Learning*)[3]. The system performs a top down search starting from the most general set of hypothesis rules within the scope of the given language bias. In a top-down fashion (where the top goal is the given set of examples) it identifies and keeps track of the general rules of an hypothesis theory that together with the background knowledge are needed to derive the examples.

The third phase is a *post-processing* phase. This takes the output of the learning system and automatically generated the revised program $T^{'}$ by re-factoring the original rules together with the new learned rules.

4 Case Study

To validate the proposed approach a well known "driver's license" case study [7], [5] has been used. We report in this section the main results, we discuss them and exemplify the revision step for one of the activities.

An event log W (containing 50 process instances) is generated simulating the execution of the *actual* process through CPN Tools [8]. We use artificial traces produced by a simulation rather than real-life logs because real-life event logs usually contain imperfections. On the contrary, by using simulation we can have more control about the properties of the event log to validate the approach under different conditions. The *actual* process is described by the Petri net shown in Fig. 3(b). This Petri net contains a loop, a duplicate task ($applyForLicense$), an invisible task (to skip $receiveLicense$ at

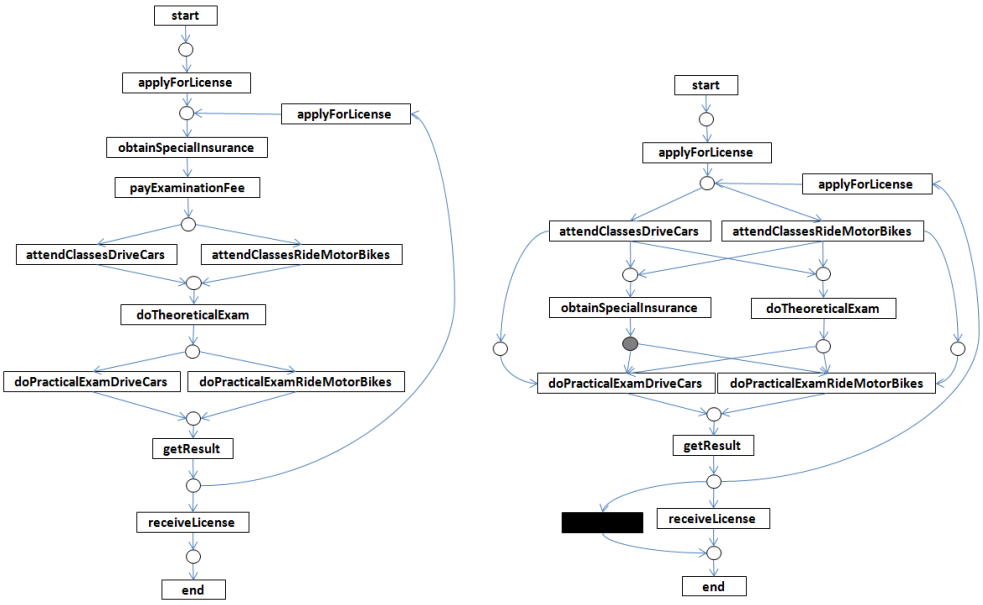

Fig. 3: (a) Existing Petri net (b) Petri Net after the revision process

the end of the process) and it is a non-free-choice Petri net because, for instance, the grey *place* in Fig. 3(b) is in the *preset*[1] of more than one transition ($doPracticalExamDriveCars$ and $doPracticalExamRideMotorBikes$) but it is not the only place in the preset of $doPracticalExamDriveCars$ neither in the preset of $doPracticalExamRideMotorBikes$.

The Petri net P, shown in Fig. 3(a), describes a currently available model of the process. The structure of this Petri net is similar to the Petri net representing the actual process. However here the non-free-choice constructs and the invisible task are missing. These constructs should be identified in the revised Petri net. Moreover in P the activities $obtainSpecialInsurance$ and $payExaminationFee$ are performed before attending the courses. In the revised Petri net $obtainSpecialInsurance$ should be performed after attending the courses and the activity $payExaminationFee$ should be deleted.

1. Translation to logical model
The first step of our approach is to formalise P as a logical model $P_{\mathcal{L}}$ using the predicate ns. $P_{\mathcal{L}}$ is obtained using the patterns shown in Fig. 2 and is shown schematically in Fig. 4(a).

[1] A preset of a transition x is the set of the places y such that there is an arc from y to x.

	activity	precondition
a	*start*	*true*
b	*applyForLicense*	*ns(a,b)*
b	*applyForLicense*	*(ns(j,b) ∧ ns(j,k)) ∧ occursLessThan(b,3)*
c	*obtainSpecialInsurance*	*ns(b,c)*
d	*payExaminationFee*	*ns(c,d)*
e	*attendClassesDriveCars*	*ns(d,e) ∧ ns(d,f)*
f	*attendClassesRideMotorBikes*	*ns(d,e) ∧ ns(d,f)*
g	*doTheoreticalExam*	*ns(e,g) ∨ ns(f,g)*
h	*doPracticalExamDriveCars*	*ns(g,h) ∧ ns(g,i)*
i	*doPracticalExamRideMotorBikes*	*ns(g,h) ∧ ns(g,i)*
j	*getResult*	*ns(h,j) ∨ ns(i,j)*
k	*receiveLicense*	*ns(j,b) ∧ ns(j,k)*
l	*end*	*ns(k,l)*

(a)

	activity	precondition
a	*start*	*true*
b	*applyForLicense*	*ns(a,b)*
b	*applyForLicense*	*(ns(j,b) ∧ ns(j,k) ∧* **ns(j,l)***) ∧ occursLessThan(b,3)*
c	*obtainSpecialInsurance*	***ns(e,c) ∨ ns(f,c)***
e	*attendClassesDriveCars*	***ns(b,e) ∧ ns(b,f)***
f	*attendClassesRideMotorBikes*	***ns(b,e) ∧ ns(b,f)***
g	*doTheoreticalExam*	*ns(e,g) ∨ ns(f,g)*
h	*doPracticalExamDriveCars*	(ns(g,h) ∧ ns(g,i)) ∧ (**ns(c,h) ∧ ns(c,i)**) ∧ **ns(e,h)**
i	*doPracticalExamRideMotorBikes*	(ns(g,h) ∧ ns(g,i)) ∧ (**ns(c,h) ∧ ns(c,i)**) ∧ **ns(f,i)**
j	*getResult*	*ns(h,j) ∨ ns(i,j)*
k	*receiveLicense*	ns(j,b) ∧ ns(j,k) ∧ **ns(j,l)**
l	*end*	ns(k,l) ∨ (**ns(j,b)** ∧ **ns(j,k)** ∧ **ns(j,l)**)

(b)

Fig. 4: (a) Obsolete model (b) Revised model

Note that $occursLessThan(b, 3)$ specifies that the activity b cannot be executed more than three times in a process instance.

2. Training data generation

In the second step the training data set $E = E^+ \cup E^-$ is generated from W. In particular approximately 600 positive examples are extracted from the event log. Starting from the positive examples AGNEs algorithm allows to generate the negative ones. In our experiment we use an injection probability $\pi = 0.2$. This means that we consider only 20% of the whole set of the generated negative examples. Approximately 100 negative examples are generated (e.g. *class(c, 1, 4, completed), class(c, 1, 9, completed), ... not class(c, 15, 11, completed)*) $class(act, t_1, t_2, completed)$ means that the activity *act* can be *completed* in the process instance t_1 at the time point t_2, since this is what happens in the log. Negative examples show the behaviours which cannot take place.

3 Revision

Starting from $P_{\mathcal{L}}$ and E the revision algorithm is executed on each activity x singularly. As previously stated, the iterative deepening implementation of TAL first checks if a solution with no revision exists, i. e. whether $B \cup P_{\mathcal{L}} \models E_x$, where E_x is the subset of E that refers to the activity x and B contains rules (1) and (2) and the definition of the *occursLessThan* predicate. This holds only for the $doTheoreticalExam$ and $getResult$ activities.

For all other activities a revision is learned. The result of the revision is the definition reported in given in Figure 4(b). The preconditions of the activities are reported in terms of the *ns* predicate and the or-split patterns are enclosed in brackets. Note that adding or deleting or-split is considered as a single elementary revision. We illustrate the partial results of the revision process for the activity $obtainSpecialInsurance$.

Example 1. Revision for the activity $obtainSpecialInsurance$. P_c refers to the rules in $P_{\mathcal{L}}$ referred to the activity c

3.1. Revision: pre-processing. In this phase, all the rules in $P_{\mathcal{L}}$ are rewritten using the *meta-predicates try* and *exception*. This transformation sets the learning task to compute exceptions cases for rules in $P_{\mathcal{L}}$ and instances of body literals that can be deleted.

$$\tilde{P}_c = \begin{cases} class(c, BId, T, completed) \leftarrow \\ \quad try(1, 1, ns(b, c)), \\ \quad \neg exception(1, class(c, BId, T, completed)) \end{cases}$$

3.2. Revision: learning. The learning phase takes as input the transformed (revisable) program $\tilde{T}$, the (unrevisable) background knowledge B, the extended language bias $\tilde{P_{\mathcal{L}}}$ and a set E of examples. It computes an inductive solution H containing information about deletions, exceptions and new rules (whenever the given language bias $L \neq \emptyset$) such that $B \cup \tilde{P_{\mathcal{L}}} \cup H \models E$ (ensured by the soundness of the ILP system deployed in our revision approach).

$$H = \begin{cases} class(c, BId, T, completed) \leftarrow \\ \quad ns(e, c) \\ class(c, BId, T, completed) \leftarrow \\ \quad ns(f, c) \\ exception(1, class(c, BId, T, completed)) \end{cases}$$

3.3. Revision: post-processing. The last phase constructs the revised theory $P'_{\mathcal{L}}$ from the output of the learning phase. This is an automatic re-factoring process that takes the revisable program $\tilde{P_{\mathcal{L}}}$ given to the learning system and the generated hypothesis H and transform them into an equivalent program $P^{'}_{\mathcal{L}}$ that represented the revised Petri net model. The transformation satisfies the property that $B \cup \tilde{P_{\mathcal{L}}} \cup H$ is equivalent to $B \cup P^{'}_{\mathcal{L}}$.

$$P_c{}' = \begin{cases} class(c, BId, T, completed) \leftarrow \\ \quad ns(e, c) \\ class(c, BId, T, completed) \leftarrow \\ \quad ns(f, c) \end{cases}$$

The learning phase generates an exception that has the effect of deleting the entire existing rule. Two new rules are learned defining an or-join.

4. Translation to Petri net. The final outcome of the learning is mapped into a model that is the actual model shown in Figure 3(b). The revision algorithm is able to tranform the free choice Petri net P in the non-free choice Petri net P'. The preconditions of the *end* activity, reveal the presence of an invisible task. In general the presented revision algorithm is able to handle all common constructs in a Petri net.

5 Conclusion

Over the last decade a variety of techniques and algorithms have been proposed for mining process models from event logs, showing that information contained in the logs can be used to formalise or improve process models. Proposed methods [12] have shown that event logs can be used to construct from scratch models underlying an automated process (i.e. process discovery), or to identify discrepancies between event logs generated by an automated process in place and a predefined process model representing its formal definition (i.e. conformance testing). Proposed approaches of process mining have mainly been focused on the realization of (customized) algorithms suitable for different aspects of process discovery, whereas approaches on conformance testing have been concerned mainly on the identification of metrics to evaluate discrepancies between existing models and actual process executions. Our approach presents various advantages with respect to the existing techniques. The main one is that it is able to learn incrementally from event logs whilst preserving as much as possible of the previously learnt model. By using our *revision through learning* approach we have shown that minimally revised process models can be learnt through an automated process. Additionally, in our approach the analyst can lock, through an explicit language bias, parts of the model he/she considers to be correct and can explore alternatives. This is facilitated by the use of an exhaustive top-down search where a solution is guaranteed to be found if one exists. Performing an exhaustive top-down search is particularly advantageous when negative execution instances are logged. In this case even a single instance may be sufficient to obtain an appropriate revision. Although not shown in this paper due to space limitations, our approach can also be used in the presence of incomplete or noisy data. Noise in the event logs can be handled through a probabilistic extension of our learning system that we are currently developing. The outcomes in this case will be a revised model that is a "best fit" to the given data. Another advantage of the approach presented in this paper is that it provides a uniform methodology and tool support for both the tasks of extraction (i.e. mining) of process models as well as revision of an existing model. Moreover the declarative representation of the business process model used as input to our learning system makes the approach flexible enough to cover different classes of process models. The same learning system can be used to discover and revise different classes of business process models, e.g. those with or without time constraints, those with or without concurrent tasks, etc. The generality of our learning approach depends on the generality of the declarative language used to formalize the business process models. So far we have shown, for simplicity, particular types of process models that include features like loops, concurrent tasks, non free-choice of constructs. The declarative representation of the process models can be appropriately extended to allow notions of time, composite events and invisible tasks, as well as any additional feature expressible in first-order logic. Our approach will be capable of extracting and revising models with some, any or all such characteristics using the same underlying learning system.

References

1. Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.
2. D. Corapi, O. Ray, A. Russo, A. Bandara, and E. Lupu. Learning rules from user behaviour. In *Artificial Intelligence Applications and Innovations III*, volume 296, pages 459–468, 2009.
3. D. Corapi, A. Russo, and E. Lupu. Inductive logic programming as abductive search. In *International Conference on Logic Programming (To Appear)*, 2010.
4. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Logic Programming*, pages 1070–1080. MIT Press, 1988.
5. Stijn Goedertier, David Martens, Jan Vanthienen, and Bart Baesens. Robust process discovery with artificial negative events. *Journal of Machine Learning Research*, 10:1305–1340, June 2009.
6. Markus Hammori, Joachim Herbst, and Niko Kleiner. Interactive workflow mining: requirements, concepts and implementation. *Data Knowl. Eng.*, 56(1):41–63, 2006.
7. A. K. Medeiros, A. J. Weijters, and W. M. Aalst. Genetic process mining: an experimental evaluation. *Data Min. Knowl. Discov.*, 14(2):245–304, 2007.
8. A. K. Alves De Medeiros and C. W. Günther. Process mining: Using cpn tools to create test logs for mining algorithms. In *Proceedings of the Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 177–190, 2005.
9. A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In C. Bussler et al., editor, *BPM 2005 Workshops (Workshop on Business Process Intelligence)*, volume 3812, pages 163–176, 2006.
10. Leon Shapiro and Ehud Y. Sterling. *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press, April 1994.
11. Wil van der Aalst, Marlon Dumas, C. Ouyang, Anne Rozinat, and H. M. W. Verbeek. Choreography conformance checking: An approach based on bpel and petri nets. In Frank Leymann, Wolfgang Reisig, Satish R. Thatte, and Wil van der Aalst, editors, *The Role of Business Processes in Service Oriented Architectures*, number 06291 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
12. Wil M. P. van der Aalst, Boudewijn F. van Dongen, Joachim Herbst, Laura Maruster, Guido Schimm, and A. J. M. M. Weijters. Workflow mining: A survey of issues and approaches. *Data Knowl. Eng.*, 47(2):237–267, 2003.
13. Boudewijn F. van Dongen, Ana Karla Alves de Medeiros, and L. Wen. Process mining: Overview and outlook of petri net discovery algorithms. *T. Petri Nets and Other Models of Concurrency*, 2:225–242, 2009.
14. James Wogulis and Michael Pazzani. A methodology for evaluating theory revision systems: Results with Audrey II. In *13th IJCAI*, pages 1128–1134, 1993.