2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

Declarative Process Discovery with Evolutionary Computing

Seppe K.L.M. vanden Broucke and Jan Vanthienen and Bart Baesens

Abstract— **The field of process mining deals with the extraction of knowledge from event logs. One task within the area of process mining entails the discovery of process models to represent real-life behavior as observed in day-to-day business activities. A large number of such process discovery algorithms have been proposed during the course of the past decade, among which techniques to mine declarative process models (e.g. Declare and AGNEs Miner) as well as evolutionary based techniques (e.g. Genetic Miner and Process Tree Miner). In this paper, we present the initial results of a newly proposed evolutionary based process discovery algorithm which aims to discover declarative process models, hence combining these two classes (declarative and genetic) of discovery techniques. To do so, we herein use a language bias similar to the one found in AGNEs Miner to allow for the conversion from a set of declarative control-flow based constraints (determining the conditions which have to be satisfied to enable to execution of an activity) to a procedural process model, i.e. a Petri net, though this language bias can be extended to include data-based constraints as well.**

I. Introduction

PROCESS MINING entails the research field which deals with the extraction of knowledge from event logs as recorded by process aware information systems. During the past decade, the field has situated itself between the areas of Business Process Management (BPM) [1] and data mining [2]. As more and more process aware information systems are implemented, an increasing amount of event logs has become available. Analysis of such event logs provide insights into the way processes take place in real life and to what degree processes deviate from a normative, prescriptive process model.

Within the process mining field, a distinction is commonly made between three prominent tasks [3]. First of all, process discovery deals with the automated construction of a process model out of an event log. Conformance checking, secondly, starts from a process model and event log and analyzes the quality of the process model in comparison with the behavior seen in the event log or the degree of deviation which occurred in real life compared to the normative model. Lastly, process enhancement also assumes the presence of an existing process model and extends this model based on additional or more fine grained data.

Without doubt, the process discovery task has received the most attention within the process mining research community, with a multitude of algorithms being proposed to this regard. In this paper, we present a new approach for an evolutionary based process discovery algorithm which is able to learn the constraints determining the execution of an activity in a declarative manner. We combine this approach with the language bias as presented in the AGNEs Miner process discovery technique [4] to enable the conversion from a set of declarative constraints to a procedural control-flow model, i.e. a Petri net.

The remainder of this paper is structured as follows. Section II provides an overview of related work. Section III briefly outlines preliminary concepts and definitions. Next, Section IV describes the evolutionary process discovery technique in detail. Section V describes the first experimental results obtained by our technique. The paper is concluded in Section VI, where possibilities for future work are also discussed.

Seppe K.L.M. vanden Broucke, Jan Vanthienen and Bart Baesens are with the Department of Decision Sciences and Information Management, KU Leuven, Naamsestraat 69, B-3000, Leuven, Belgium (email: {seppe.vandenbroucke,jan.vanthienen,bart.baesens}@kuleuven.be).

Bart Baesens is also associated to the School of Management, University of Southampton, Highfield Southampton, SO17 1BJ, United Kingdom.

We thank the KU Leuven research council for financial support under grand OT/10/010 and the Flemish Research Council for financial support under Odysseus grant B.0915.09.

II. Related Work

In the past years, many process discovery techniques have been proposed. Process discovery can be seen as an application of the machine learning of grammars [4], where the learning task can be formulated as follows: “given a sequence database that contains a finite number of sequence, extract a generative model that describes its spatio-temporal properties.” The foundational approaches to process discovery were formulated by Agrawal et al. [5], Cook and Wolf [6], and Lyytinen et al. [7].

The α-algorithm can be considered as one of the most substantial techniques in the process mining field. Van der Aalst et al. [8] prove that it can learn structured workflow nets from complete event logs (with respect to all allowable binary sequences), assuming that the event log does not contain any noise. Therefore, the α-algorithm is sensitive to noise and incompleteness of event logs. Moreover, the original α-algorithm is incapable of discovering short loops or non-local, non-free choice constructs. Alves de Medeiros et al. [9] improved the original α-algorithm to mine short loops and named it α+. Other techniques have also been proposed to improve the original α-algorithm. Wen et al. [10] for instance propose the α++-algorithm, capable to detect non-free choice constructs. The same authors have also proposed to take advantage of both “start” and “complete” event types in order to detect concurrency.

In order to remedy the robustness problem of the α-algorithms, Weijters et al. developed Heuristics Miner [11]. This process discovery technique extends the formal α-algorithm by applying frequency information with regard to relationships between activities in an event log. Heuristics Miner can discover short loops and non-local dependencies, but lacks to detect duplicate activities. In 2010, Burattin and Sperduti proposed an adaption of the Heuristics Miner algorithm, named Heuristics Miner++ [12], which extends the former by considering activities with time intervals, i.e. having a starting and ending time instead of being logged as an atomic, zero-duration event. The same authors have also proposed a modified Heuristics Miner which is able to deal with streaming event data [13]. Weijters and Ribeiro have also created a modified version of their Heuristics Miner algorithm, denoted as Flexible Heuristics Miner [14], which outputs the mined model as a Causal net. The Flexible Heuristics Miner also incorporates a new strategy to mine split/join semantics, based on the frequencies each input and output binding is activated for each task.

Following closely after the inception of Heuristics Miner, Günther and van der Aalst developed another heuristic process discovery technique, Fuzzy Miner [15], geared specifically towards mining unstructured, spaghetti like processes. Just as Heuristics Miner, this technique does not mine duplicate or invisible activities. Users can apply a number of filters to create higher-level abstractions of complex process models, either by filtering out activity nodes or edges, or by clustering groups of activities in the mined model. Although this approach enables the robust exploration of event logs, it suffers from the drawback that a Fuzzy model cannot be translated to a formal Petri net which limits a comparative evaluation to other process discovery techniques [16].

Scholars have also proposed to use machine learning techniques in the context of process discovery. Maruster et al. [17] investigate the use of rule-induction to predict dependency relationships between business process activities, by using a propositional rule induction technique (RIPPER) on a table of direct metrics for each process task in relation to each other process task. Ferreira and Ferreira [18] apply a combination of ILP learning and partial-order planning techniques to process mining. Lamma et al. [19] also describe the use of ILP to process mining. The authors assume the presence of negative sequences to guide the search algorithm, unlike the approach of Ferreira and Ferreira, who use partial-order planning to present the user with an execution plan to accept or reject (a negative example), this approach does not provide an immediate answer to the origin of such negative examples, or: "negative events." AGNEs Miner proposed by Goedertier et al. [4] addresses the difficulty by representing the discovery task as first-order classification learning on event logs, supplemented with artificially generated negative events. The process discovery algorithm then learns the discriminating conditions that determine whether an activity can take place or not. Another approach based on machine learning theory was proposed by Greco et al. This technique, called DWS mining, can be described as a hierarchical and iterative procedure that refines the process model in each step, based on clustering of patterns sharing similar behavior [20]. This approach guarantees full compliance with the event log and increasingly improves the precision of the process model. The authors later also introduce AWS mining [21], consisting of both mining and abstraction algorithms geared to build a tree-like schema. The idea to represent mined process models through different views at different levels of abstraction is also the driver behind FSM Miner/Petrify, a process discovery technique proposed by van der Aalst et al. [22], which constructs a transition system from the traces in an event log which is then synthesized by means of theory of regions to a Petri net. A similar technique is described by Carmona et al. [23]. Another algorithm, ILP Miner [24], entails the application of Integer Linear Programming (ILP) to process discovery. This technique is also based on the well-known theory of regions.

Due to the limitations of local search, early approaches to process discovery were generally not able to discover complex constructs such as non-free choice, invisible tasks and duplicate tasks. Therefore, Alves de Medeiros et al. [25] were the first to apply a genetic algorithm for process discovery so as to benefit from global search. Their Genetic Miner defines its search space in terms of causal matrices. Because of the global search property, Genetic Miner is capable of detecting non-local patterns in the event log. Buijs et al. have also proposed an evolutionary algorithm towards process discovery [26]. Contrary to earlier approaches, the authors aim to discovery another process representation—Process Trees—which guarantees the discovery of sound Petri nets. The main drawback of this technique lies in the time-intensive step of performing the fitness evaluation.

In this paper, we propose a new approach towards evolutionary process discovery. Contrary to earlier techniques, we aim to discover a *declarative* process model, by deriving the constraints that determine whether an event can take place or not, given a history of events of other activities. However, we combine this approach with the declarative language bias as outlined in AGNEs Miner [4] so that the set of preconditions for each activity can be converted to a Petri net.

III. Preliminaries

This section outlines some preliminary concepts and definitions which will be utilized in the remainder of the paper.

A. Event Logs

Process discovery algorithms start from a so-called event log and output a process model using a particular representational language. An event log consists of events that pertain to process instances: executions of a process in a system. A process instance is a grouping of activities whose state changes are recorded as events. In order to obtain a usable event log, it is assumed that it is possible to record events so that each event refers to an activity (e.g. "sign order"), the process instance (e.g. "PI101") and that the events are ordered, either based on a time stamp (e.g. "2012-11-10 09:08:07") or on the basis of relative ordering (a sequence number). In some cases, the specific state transition of the activity is also recorded in the event, for example to denote both when an activity was started ("start" activity transition) versus its time of completion ("complete" activity transition).

We will make use of the following notations. Let event log L be defined as a multiset of traces (process instances). The cardinality (or size) of an event log $|L|$ denotes the total number of traces in the log, including duplicates. A trace $\sigma \in L$ is a finite sequence of events with length $|\sigma|$ and with σ_i the event at position i in trace σ. Since an ordering is explicitly defined between events in a sequence and the related process instance can be left implicit, the events themselves can simple be denoted based on their activity name, e.g. $\sigma = \langle a, b, c, d \rangle$. The set of activities occurring in the event log is then denoted as $A = \{\sigma_i | \sigma \in L, i = 1 \ldots |\sigma|\}$.

B. Petri nets

Our proposed process discovery algorithm attempts to derive constraining pre-conditions determining whether an activity can take place or not, but since we will ultimately convert the set of mined constrained to a Petri net, we provide the definition of a Petri net as follows. A Petri net is a triplet $PN = (P, T, F)$ with P a finite set of places and T a finite set of transitions with $P \cap T = \emptyset$. F is the set of flows $F \subseteq (P \times T) \cup (T \times P)$. The state of a Petri net is defined by its marking $M : P \rightarrow \mathbb{N}_0$. A transition is enabled in a given marking whenever all of its input places contain at least one token. Firing a transition then consumes a token from each input place and produces a token in each output place [27].

C. Control-flow Based Declarative Constraints

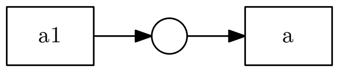
The general learning task consists of predicting for a given activity $a \in A$ whether it is possible to execute this activity at a given time point (or position) in a given historical execution sequence $\sigma \in L$.

To learn the preconditions for each of the activities in a given event log, we apply a similar language bias as the one utilized in AGNEs Miner [4], as the goal (here) remains to construct a graphical process model from a given event log. As such, we re-use the logical "no-sequel" predicate, $\mathit{NS}(a_1, a, \sigma, t)$ with $a_1, a \in A$, $\sigma \in L$ and t the position (or time) of observation. The *NS* predicate can thus be defined as follows:

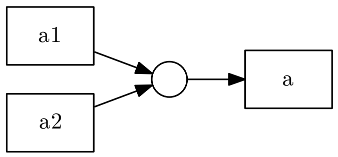
$$\begin{aligned}\forall a_1, a \in A, \sigma \in L, t \in [1, |\sigma|] : [\exists \sigma_i \in \sigma : [\sigma_i = a_1 \wedge i < t] \\ \wedge \nexists \sigma_j \in \sigma : [\sigma_j = a \wedge i < j < t]] \\ \Rightarrow \mathit{NS}(a_1, a, \sigma, t)\end{aligned}$$

The predicate $\mathit{NS}(a_1, a, \sigma, t)$ evaluates to true when at the time of observation t, an activity a_1 has been completed but not yet followed by an activity a. In AGNEs Miner, these predicates are learnt per trace (σ) and per time of observation (t), but are then left implicit to create the actual activity preconditions, as the resulting process model should hold in all traces at all times of observation. Since we will not derive these predicates from the event log but immediately use the same language bias as a declarative modeling construct, we also leave the arguments σ and t implicit and will thus denote the predicate simply as: $\mathit{NS}(a_1, a)$.

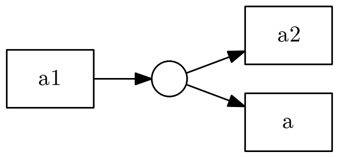
By creating conjunctions and disjunctions of the *NS* predicate, this construct enables to learn fragments of Petri nets. Fig. 1 shows the Petri net patterns which can be learnt in this manner. Note that parallelism (i.e. AND join/splits) are not represented by a single conjunction or disjunction of $\mathit{NS}(a_1, a)$ predicates (i.e. $\mathit{NS}(a_1, a) \wedge \mathit{NS}(a_1, a_2)$ for an activity a denotes a XOR split) but rather by the number of such groups present in the preconditions for an activity (i.e. $\mathit{NS}(a_1, a) \wedge \mathit{NS}(a_2, a)$ for an activity a denotes an AND join). Section IV provides more details on the conversion to a Petri net.



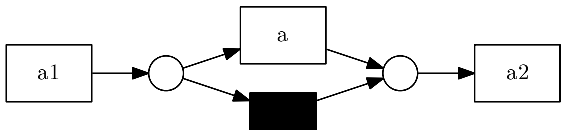
Sequence pattern: $\mathit{NS}(a_1, a)$.



XOR join pattern: $\mathit{NS}(a_1, a) \vee \mathit{NS}(a_2, a)$.



XOR split pattern: $\mathit{NS}(a_1, a) \wedge \mathit{NS}(a_1, a_2)$
(same precondition for a_2).



Skip pattern: $\mathit{NS}(a_1, a) \wedge \mathit{NS}(a_1, a_2)$
(precondition for a_2: $\mathit{NS}(a_1, a_2)$).

Fig. 1. The different Petri net patterns which can be modelled by the *NS* predicates [4].

Given the definition of the $\mathit{NS}(a_1, a)$ predicate and the control-flow patterns it enables, we now present our evolutionary approach in order to discover the set of constraints for a set of given activities. Our approach differs from AGNEs Miner in the sense that no information about negative events is required during the discovery phase, and that an evolutionary technique is applied, rather than Inductive Logic

Programming, which resolves some of the drawbacks of this earlier approach (such as the limited length of disjunctions and conjunctions). Our approach differs from other evolutionary process discovery algorithm since we do not define our solution structure in terms of a procedural process model, but rather as a set of declarative constraints.

IV. Methodology

A. General Overview

To implement our evolutionary algorithm, we utilize the Watchmaker Java library [28], an extensible, high-performance, open-source, object-oriented framework for implementing platform-independent evolutionary algorithms in Java. As with any genetic algorithm, the general flow of our approach can be summarized as follows. First, an initial population of solutions with a given size is constructed (*genesis*) using either blank, randomized, or other genesis operator. Next, the algorithm loops through different "generations" in which the following steps are performed. First, a *fitness evaluation* is executed to score each member of the population. Based on this ranking, population members are selected (*selection*) to produce the "offspring" for the following generation. The creation of such offspring is performed by applying *crossover* operators on the selected population members. Finally, once a new generation is constructed, a series of *mutation* operators is applied in order to introduce variety in the solution space covered by the population (this step serves as a way to escape from local optima). The algorithm stops whenever a certain fitness or number of generations threshold is reached.

In the following subsections, we describe each of the steps mentioned above in more detail.

B. Solution Representation

We represent solutions as a rule base of preconditions R with the set of preconditions for an activity $a \in A$ represented as $R^a \in R$. Our approach allows one to learn multiple sets of preconditions for an activity type, $\bigvee (R_i^a)$ (with $i = 1, \ldots, |R^a|$) which correspond to duplicate transitions once the conversion to a Petri net is performed. Each particular set of precondition R_i^a is defined as a conjunction of groups of *NS* predicates: $R_i^a = \bigwedge (R_{i,j}^a)$ (with $j = 1, \ldots, |R_i^a|$). Each such group will be converted to an input place in the resulting Petri net. The *NS* predicates in each group $R_{i,j}^a$ are either in a conjunctive or disjunctive relation (see Fig. 1), representing XOR splits and joins respectively. When only one *NS* predicate is present in a group, a sequence is modelled. When a disjunction ($\vee$) of *NS* predicates is present in a group, they share the same second argument and model a XOR join (see Fig. 1). When a conjunction ($\wedge$) of *NS* predicates is present in a group, they model a XOR split or skip patternm and shere the same second argument. Fig. 2 provides an example of a rule base of preconditions and its respective Petri net representation.

Converting a set of preconditions to a Petri net is a relatively trivial operation. For each set of preconditions

$$
\begin{aligned}
&a :\text{-}\ true.\\
&b :\text{-}\ NS(a,b) \wedge NS(a,c).\\
&c :\text{-}\ NS(a,b) \wedge NS(a,c).\\
&d :\text{-}\ NS(a,d) \wedge NS(a,f).\\
&e :\text{-}\ NS(b,e) \vee NS(c,e).\\
&f :\text{-}\ NS(d,f).\\
&g :\text{-}\ (NS(e,g)) \bigwedge (NS(f,g)).
\end{aligned}
$$

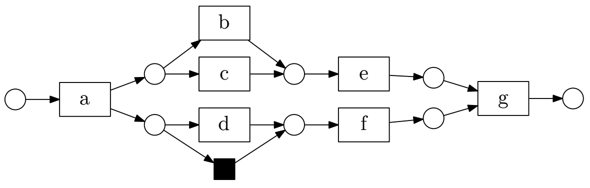


Fig. 2. Example of a set of activity preconditions and conversion to Petri net.

$R_i^a \in R^a$, a Petri net transition is constructed. Next, for each group of *NS* predicates $R_{i,j}^a \in R_i^a$, an input place for the transition at hand is added and connected. Once this is done, all *NS* predicate groups are reiterated to construct the Petri net patterns as shown in Fig. 1. Finally, source and sink places are added and connected.

A pruning step was implemented to prune the preconditions for an activity. Empty *NS* groups are purged, and duplicate activities, XOR joins and XOR splits are removed if they are subsumed by a more specific construct. This pruning step is applied during each generation of the evolutionary algorithm on the whole population.

C. Genesis Operators

To construct an initial population, two genesis operators are implemented. The first one constructs population members with no preconditions for each activity type, so that each activity is free to execute at any point in time. In terms of process model quality dimensions, this compares to models having a perfect recall, but low precision score. This trivial method to construct fitting process models emphasizes the need to incorporate both recall and precision in the evaluation of solutions, which we will touch upon again below. The second genesis operator constructs random solutions where the maximum values for $|R^a|$, $|R_i^a|$ and $|R_{i,j}^a|$ can be provided.

An important remark entails the way groups of *NS* predicates $R_{i,j}^a$ are changed or modified during the execution of the evolutionary algorithm. To enable the valid construction of a set of preconditions, each group of *NS* predicates needs to adhere to the following conditions:

- *NS* predicates belonging to groups $R_{i,j}^a$ containing one *NS* predicate or a disjunction of multiple *NS* predicates (sequence or XOR join) should be of the form $NS(a_i, a)$;

- *NS* predicates belonging to groups $R^a_{i,j}$ containing a conjunction of *NS* predicates (XOR split or skip) should contain one predicate of the form $NS(a_1, a)$ with the remaining predicates of the form $NS(a_1, a_2)$.

All genesis, crossover and mutation operators ensure that these conditions are adhered to.

D. Crossover Operator

One crossover operator is implemented in our evolutionary approach. This operator swaps a set number (the amount of "crossover points") of randomly chosen groups of *NS* predicates (i.e. $R^a_{i,j}$ with i and j random) between the parents to create two new solution members, $\forall a \in A$.

E. Mutation Operators

Ten mutation operators are executed by the evolutionary algorithm. Two of them deal with randomly adding and removing a rule base for a duplicate activity, i.e. adding or removing an $R^a_i \in R^a$. The next two operators randomly add or remove a group of *NS* predicates, i.e. add or remove an $R^a_{i,j} \in R^a_i$ and thus change existing solutions at a more granular level. The next two operators are defined on the most granular level, and add or remove random *NS* predicates to existing *NS* groups. Next, two mutation operators are included which change a random *NS* predicate, or change the a_1 argument in a conjunctive group of *NS* predicates $NS(a_1, a_2)$ to another activity.

Another mutation operator switches the type of a group of *NS* predicates (i.e. from conjunction to disjunction and vice versa). To change a group of conjunctive predicates $NS(a_1, a_2) \wedge NS(a_1, a_3) \wedge \cdots \wedge NS(a_1, a)$ to a disjunction, a predicate is added for the shared first argument of the conjunctive group, together with predicates for each second argument of the conjunctive group different from the activity under consideration, so that the new disjunctive group reads as: $NS(a_1, a) \vee NS(a_2, a) \vee \cdots \vee NS(a_3, a)$. Changing a disjunction to a conjunction is performed similarly, with one random activity from the first arguments of the predicates in the disjunctions chosen to become the shared first argument in the conjunction.

The final mutation operator completely discards a solution and generates a new population member in its stead.

F. Fitness Evaluation

A first straightforward manner to evaluate the fitness of population members is to convert each solution to a Petri net and apply a well-known process model quality metric, such as the alignment based fitness/precision metrics [29] or the Weighted Behavioral Recall/Precision metrics [30][1].

However, since both the conversion to a Petri net and the calculation of the aforementioned metrics are a time intensive procedure, we also implement an additional scoring procedure which is directly applied on the collection of activity preconditions. For each activity in each trace in the event log, we evaluate whether the current set of preconditions indeed allows the execution of this activity given its history. As such, we can derive a fitness metric f as follows:

$$f = \frac{|\{\sigma_i | \sigma_i \in \sigma, \sigma \in L : \mathit{correct}(\sigma_i, \sigma, R^{\sigma_i})\}|}{|\{\sigma_i | \sigma_i \in \sigma, \sigma \in L\}|}$$

with $correct(\sigma_i, \sigma, R^{\sigma_i})$ a function denoting whether an activity can execute given a trace history and a set of preconditions, meaning that there $\exists R^{\sigma_i}_x \in R^{\sigma_i}$ which is satisfied at the current position of observation. Note that scoring population members in terms of fitness alone does not suffice, as this would lead to the derivation of trivial solutions where each activity is free to be executed at any point in time. As such, we also define a precision metric p as follows:

$$p = \frac{|\{n | n \in \mathit{NE}(\sigma_i), \sigma_i \in \sigma, \sigma \in L : \neg \mathit{correct}(n, \sigma, R^{\sigma_i})\}|}{|\{n \in \mathit{NE}(\sigma_i), \sigma_i | \sigma_i \in \sigma, \sigma \in L\}|}$$

with the $\mathit{NE}(\sigma_i)$ function returning a set of so called "negative events", meaning activities which should be prohibited from being executed given the execution history before σ_i. Such set of negative events can be derived in linear time from a given event log by utilizing an artificial negative event generation technique we have described in [30].

Finally, for both of the above fitness evaluation alternatives, a solution's global score is then calculated as the F-measure [32], i.e. the harmonic mean between fitness and precision: $F = 2\left(\frac{p \times f}{p + f}\right)$.

V. Experimental Evaluation

This section discusses the first experimental results of the proposed genetic algorithm. We apply the genetic mining algorithm on the *l2l* event log which has also been utilized in earlier benchmarks by Alves de Medeiros et al. in the context of their Genetics Miner [33]. The "true" process model of this event log is depicted in Fig. 3. In addition, we have also performed experiments on the three event logs utilized in [26] to test the genetic Process Tree Miner: event log *seq* contains a single trace: $\langle a, b, c, d, e, f\rangle$, event log *xor* contains six activities in an exclusive choice: $L = \{\langle a\rangle, \langle b\rangle, \langle c\rangle, \langle d\rangle, \langle e\rangle, \langle f\rangle\}$ and event log *and* contains all 720 possible permutations of the six activities, and thus described the model where these activities are executed in parallel.

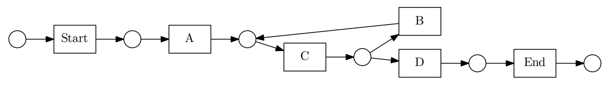


Fig. 3. The underlying model behind the *l2l* event log.

[1]Note that the use of the the term "fitness" here both applies to the overall score of a population member in the evolutionary pool and to one of the four quality dimensions relating to the quality of process models [31], which is sometimes also denoted with "recall". To avoid confusion, we will henceforth utilize the term "score" to indicate a solution's overall evolution fitness score.

The configuration parameters of the genetic algorithm are as follows:

- Population size: 30;
- Genesis operator: randomized solution, no duplicate tasks, one or two (random choice) conjunctions R_i^a, one or two (random choice) *NS* groups per conjunction $R_{i,j}^a$;
- Crossover enabled with 1 crossover point;
- All mutation operators enabled with default occurrence chances;
- Fitness evaluation: harmonic mean between Weighted Behavioral Recall/Precision metric calculated on the converted Petri net.

We compare the performance of our genetic algorithm to the case where solutions are generated randomly (using the same randomized genesis operator as used by the genetic algorithm to construct the initial population). Table I shows the result of this experiment after generating ten thousand randomized solution. For none of the logs, a solution with perfect fitness could be obtained (the 0 fitness for the *xor* log is due to (i) the fact that none of the randomized solutions could be converted to a Petri net with a unique source place, and (ii) traces in this log only containing a single event, leading to a recall score of zero).

TABLE I
AVERAGE SOLUTION SCORE WITH CONFIDENCE INTERVAL AFTER GENERATING TEN THOUSAND RANDOMIZED SOLUTIONS.

Event Log:	***seq***	***xor***	***and***	***permu***
Avg. Fitness:	0.28	0.00	0.54	0.25
Conf. Int. (95%):	[0.09, 0.47]	[0.00, 0.00]	[0.37, 0.70]	[0.09, 0.42]

Next, we run our evolutionary algorithm on the four event logs. Table II shows the results obtained after running the experiment. For the *seq* event log, we obtain a perfectly fitting model, albeit containing a superfluous AND construct which does not impact the model's quality regarding fitness and precision, but is less straightforward to interpret than the model just modeling the sequence as is. Incorporating simplicity-based criteria in the evaluation of solution candidates is thus a possible avenue for further improvement.

The *xor* and *l2l* logs are mined correctly. However, for the *and* event log, no suitable model could be found. The evolutionary algorithm attempts to improve the model by introducing more invisible and duplicate activities and entangling them by connecting them together, instead of reverting to the situation where all activities are included in a single AND split. Fine-tuning the mutation operators helps to resolve this issue, but as this requires user-intervention, we do not report those in Table II. Adapting the parameters of the evolutionary algorithm is thus also left for future work.

We conclude that our experiment clearly shows an improvement over the random case. As such, we believe that this technique forms an interesting approach towards process discovery.

VI. Discussion and Conclusion

This paper presented a new outline for an evolutionary based process discovery algorithm which is able to learn the constraints determining the execution of an activity in a declarative manner. To do so, we utilize the language bias from AGNEs Miner in order to enable to conversion from a set of declarative constraints to a procedural control flow model. Initial results of our technique prove promising, although various possibilities towards future work exist, which we discuss in the following paragraphs.

A first possibility towards future work entails the incorporation of declarative constraints other than the control-flow based constraints as modeled by the *NS* predicate. For example, it is possible to incorporate conditions governing the maximum amount of times a loop can be executed, or incorporate data-based value checks so that activity preconditions model that a certain activity can only be executed when certain data-based properties are satisfied. (I.e. an activity perform exhaustive claim check in an insurance process is only executed when the claim amount is higher than a certain number.) Initial tests indeed support this idea. Fig. 4 for example shows the *l2l* process model where the evolutionary algorithm has learned an extra precondition for activity B based on the number of occurrences of B already observed in the trace, which is converted to a transition guard in the resulting Petri net. It is also possible to construct a declarative model which only incorporates a minimal required set of control-flow based constraints, for example by incorporating the Declare rule templates.

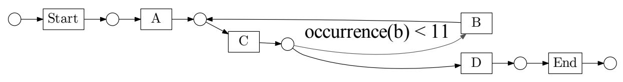


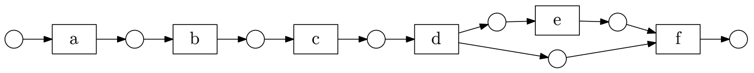
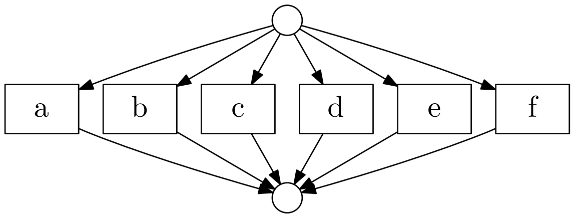
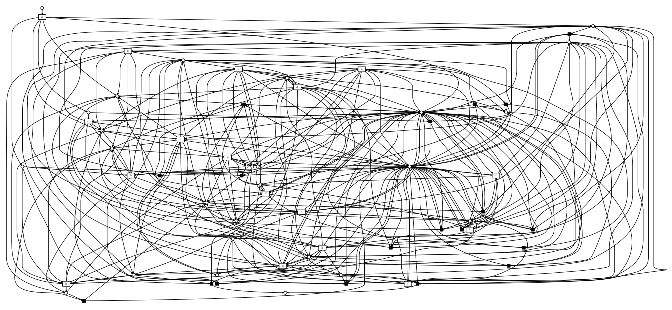
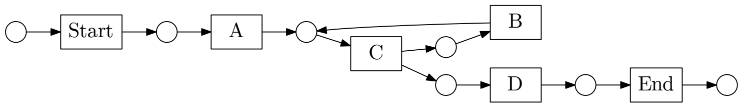
Fig. 4. Model learnt for the *l2l* event log with an extra occurrence based precondition for B.

Second, possibilities exist to modify the population scoring function. A first way to do so entails the incorporation of generalization and simplicity quality dimension based metrics [31]. Second, since the preconditions for each activity are learned independently, it is possible to "guide" the evolutionary algorithm in a more focused manner by preventing certain modifications to preconditions of a certain activity once it is determined that these fittingly describe the behavior as observed in a given event log for this activity. Initial tests performed indeed seem to indicate that this is a promising avenue to speed up and improve the performance of the evolutionary algorithm.

Finally, we also plan to incorporate more experiments in future work to accurately compare our proposed technique to other process discovery algorithms.

TABLE II

SOLUTIONS OBTAINED BY THE GENETIC MINER FOR THE FOUR EVENT LOGS INCLUDED IN THE EXPERIMENT.

Event Log	**Solution**	**Comments**
seq		Fitness: 1.00 Generations: 153 Time taken: 2m13s (Superfluous AND construct)
xor		Fitness: 1.00 Generations: 7 Time taken: 6s (Perfect model)
and		Fitness: 0.86 Generations: 200 (aborted) Time taken: 5m42s
l2l		Fitness: 1.00 Generations: 37 Time taken: 43s (Perfect model)

References

[1] J. vom Brocke and M. Rosemann, *Handbook on Business Process Management: Strategic Alignment, Governance, People and Culture*. Springer, 2010.

[2] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Addison-Wesley, 2005.

[3] W. van der Aalst, *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[4] S. Goedertier, D. Martens, J. Vanthienen, and B. Baesens, "Robust Process Discovery with Artificial Negative Events," *Journal of Machine Learning Research*, vol. 10, pp. 1305–1340, 2009.

[5] R. Agrawal, D. Gunopulos, and F. Leymann, "Mining process models from workflow logs," in *Proceedings of the 6th International Conference on Extending Database Technology (EDBT'98), volume 1377 of Lecture Notes in Computer Science*, pp. 467–483, Springer, 1998.

[6] J. Cook and A. Wolf, "Discovering models of software processes from event-based data," *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 3, pp. 215–249, 1998.

[7] K. Lyytinen, L. Mathiassen, J. Ropponen, and A. Datta, "Automating the discovery of as-is business process models: Probabilistic and algorithmic approaches," *Information Systems Research*, vol. 9, no. 3, pp. 275–301, 1998.

[8] W. van der Aalst, A. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 9, pp. 1128–1142, 2004.

[9] A. Alves de Medeiros, B. van Dongen, W. van der Aalst, and A. Weijters, "Process mining: Extending the alpha-algorithm to mine short loops," BETA working paper series 113, TU Eindhoven, 2004.

[10] L. Wen, W. van der Aalst, J. Wang, and J. Sun, "Mining process models with non-free-choice constructs," *Data Mining and Knowledge Discovery*, vol. 15, no. 2, pp. 145–180, 2007.

[11] A. Weijters, W. van der Aalst, and A. de Medeiros, "Process mining with the heuristics miner-algorithm." 2006.

[12] A. Burattin and A. Sperduti, "Heuristics miner for time intervals," in *ESANN*, 2010.

[13] A. Burattin and A. Sperduti, "Automatic determination of parameters' values for heuristics miner++," in *IEEE Congress on Evolutionary Computation*, pp. 1–8, IEEE, 2010.

[14] A. Weijters and J. Ribeiro, "Flexible heuristics miner (fhm)," in *CIDM*, pp. 310–317, IEEE, 2011.

[15] C. Günther, *Process Mining in Flexible Environments*. PhD thesis, TU Eindhoven, 2009.

[16] J. De Weerdt, M. De Backer, J. Vanthienen, and B. Baesens, "A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs," *Information Systems*, vol. 37, no. 7, pp. 654–676, 2012.

[17] L. Maruster, A. Weijters, W. van der Aalst, and A. van den Bosch, "A rule-based approach for process discovery: Dealing with noise and imbalance in process logs.," *Data Mining and Knowledge Discovery*, vol. 13, no. 1, pp. 67–87, 2006.

[18] H. Ferreira and D. R. Ferreira, "An integrated life cycle for workflow management based on learning and planning," *International Journal of Cooperative Information Systems*, vol. 15, no. 4, pp. 485–505, 2006.

[19] E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari, "Inducing declarative logic-based models from labeled traces," *Business Process Management*, pp. 344–359, 2007.

[20] G. Greco, A. Guzzo, L. Pontieri, and D. Saccà, "Discovering expressive process models by clustering log traces," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 8, pp. 1010–1027, 2006.

[21] G. Greco, A. Guzzo, and L. Pontieri, "Mining taxonomies of process models," *Data Knowl. Eng.*, vol. 67, no. 1, pp. 74–102, 2008.

[22] W. Aalst, V. Rubin, H. Verbeek, B. Dongen, E. Kindler, and C. Gnther, "Process mining: a two-step approach to balance between underfitting and overfitting," *Software and Systems Modeling*, vol. 9, no. 1, pp. 87–111, 2010.

[23] J. Carmona, J. Cortadella, and M. Kishinevsky, "New region-based algorithms for deriving bounded petri nets," *IEEE Trans. Comput.*, vol. 59, no. 3, pp. 371–384, 2010.

[24] J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik, "Process discovery using integer linear programming," *Fundam. Inform.*, vol. 94, no. 3-4, pp. 387–412, 2009.

[25] A. de Medeiros, A. Weijters, and W. van der Aalst, "Genetic process mining: an experimental evaluation," *Data Mining and Knowledge Discovery*, vol. 14, no. 2, pp. 245–304, 2007.

[26] W. Aalst, J. Buijs, and B. Dongen, "Towards improving the representational bias of process mining," in *Data-Driven Process Discovery and Analysis* (K. Aberer, E. Damiani, and T. Dillon, eds.), vol. 116 of *Lecture Notes in Business Information Processing*, pp. 39–54, Springer Berlin Heidelberg, 2012.

[27] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[28] D. Dyer, "Watchmaker Framework," 2010.

[29] A. Adriansyah, B. van Dongen, and W. van der Aalst, "Towards Robust Conformance Checking," in *BPM 2010 International Workshops and Education Track (BPM 2010), volume 66 of Lecture Notes in Business Information Processing*, pp. 122–133, 2011.

[30] S. K. vanden Broucke, J. D. Weerdt, J. Vanthienen, and B. Baesens, "Determining process model precision and generalization with weighted artificial negative events," *IEEE Transactions on Knowledge and Data Engineering*, vol. 99, no. PrePrints, p. 1, 2013.

[31] A. Rozinat, A. K. A. de Medeiros, C. W. Günther, A. J. M. M. Weijters, and W. M. P. van der Aalst, "The need for a process mining evaluation framework in research and practice," in *Business Process Management Workshops* (A. H. M. ter Hofstede, B. Benatallah, and H.-Y. Paik, eds.), vol. 4928 of *Lecture Notes in Computer Science*, pp. 84–89, Springer, 2007.

[32] J. De Weerdt, M. De Backer, J. Vanthienen, and B. Baesens, "A robust f-measure for evaluating discovered process models," in *IEEE Symposium Series in Computational Intelligence*, 2011.

[33] A. Alves de Medeiros, *Genetic Process Mining*. PhD thesis, TU Eindhoven, 2006.