Improved Artificial Negative Event Generation to Enhance Process Event Logs

Seppe K.L.M. vanden Broucke, Jochen De Weerdt,
Bart Baesens, and Jan Vanthienen

Department of Decision Sciences and Information Management, KU Leuven,
University of Leuven, Naamsestraat 69, B-3000, Leuven, Belgium
firstname.lastname@econ.kuleuven.be

Abstract. Process mining is the research area that is concerned with knowledge discovery from event logs. Process mining faces notable difficulties. One is that process mining is commonly limited to the harder setting of unsupervised learning, since negative information about state transitions that were prevented from taking place (i.e. negative events) is often unavailable in real-life event logs. We propose a method to enhance process event logs with artificially generated negative events, striving towards the induction of a set of negative examples that is both correct (containing no false negative events) and complete (containing all, non-trivial negative events). Such generated sets of negative events can advantageously be applied for discovery and evaluation purposes, and in auditing and compliance settings.

Keywords: process mining, process discovery, event logs, negative events.

1 Introduction

Many of today's organizations are currently confronted with an information paradox: the more business processes are automated, the harder it becomes to understand and monitor them. While information support systems such as Enterprise Resource planners (ERP) and modern Workflow Management systems (WfMS) provide some analysis and visualization tools in order to monitor and inspect processes – often based on key performance indicators, an abundance of data about the way people conduct day-to-day practices still remains untapped and concealed in so called event logs, capturing exactly which business activities happened at certain moments in time. The research area that is concerned with deep knowledge discovery from event logs is called process mining [1] and is often situated at the intersection of the fields of data mining and Business Process Management (BPM). Most of the attention in the process mining literature has been given to process discovery techniques [2], which focus specifically on the extraction of control-flow models from event logs [2, 3, 4, 5, 6]. One of the particular difficulties process discovery is faced with is that the learning task is commonly limited to the harder setting of unsupervised learning, since information about state transitions (e.g. starting, completing) that were prevented from

J. Ralyté et al. (Eds.): CAiSE 2012, LNCS 7328, pp. 254–269, 2012.

taking place (i.e. negative events) is often unavailable in real-life event logs and consequently cannot guide the discovery task [7]. In addition, process models frequently display complex structural behavior such as non-free choice, invisible activities (which were executed but not recorded in an event log) and duplicate activities, which make the hypothesis space of process mining algorithms harder to navigate. A third difficulty is the presence of noise in event logs, which often leads to the discovery of models which overfit the given log. In this paper, we focus our attention towards the first difficulty, namely the absence of negative examples in event logs.

Generating a robust set of negative events boils down to finding an optimal set of negative examples under the counteracting objectives of correctness and completeness. Correctness implies that the generation of false negative events has to be prevented, while completeness entails the induction of "non-trivial" negative events, that is, negative events which are based on constraints imposed by complex structural behavior, such as non-free choice constructs. The existence of the trade-off between these two goals is due to a "completeness assumption" one has to make over a given event log when generating artificial negative events. Under its most strict form, the completeness assumption states that the given event log contains all possible traces that can occur. Without some assumption regarding the completeness of a given event log, it would be impossible to induce any negative events at all, since no single candidate negative event can be introduced in the knowledge that the given log does not cover all possible cases. Note that process discovery algorithms make a similar assumption, in order to derive models which are not overly general.

In this paper, we refine an artificial negative event generation method first introduced in [7]. In the original version of this algorithm, a configurable completeness assumption is defined by proposing a window size parameter and a negative event injection probability. We aim to make the completeness assumption more configurable and the generation procedure more robust so that the induction of false negative events in cases where an event log does not capture all possible behavior is prevented (correctness), while also remaining able to derive "non-trivial" negative events, that is, negative events following from complex structural behavior (completeness).

Enhancing a process log with a correct and complete set of negative events as shown hereafter can prove beneficial for a number of reasons. First, supervised learners can now be employed in order to perform a process discovery task. Multi-relational learning techniques have already been applied in this context, using inductive logic programming in order to learn a process model from a given event log supplemented with negative events. We refer to [8, 9, 10, 11, 12] for a detailed overview of inductive logic programming and its applications in the field of process mining. Second, event logs supplemented with negative events can also be applied towards evaluation purposes, in order to assess fitness and precision of process discovery algorithms. For instance, the metrics in [7, 13] make use of event logs containing negative events to compose a confusion matrix in concordance with standard metric definitions in the field of data mining.

Third, since negative events capture which actions in a process could not occur, compliance and conformance related analysis tasks present themselves as a natural area of application for negative events. For example, auditors can cross-check a set of induced negative events with the expected behavior of real-life processes to determine if prohibited actions were indeed captured in this generated set of rejected activity state transitions. Another specific example is access rule mining. Often, users are not so much interested in the actual control policy already present in a specific process, as this policy might already have been formally specified, but rather in a more restrictive policy that reveals which access rules are unnecessary. Negative events can then stipulate that a particular agent did never perform a particular activity at a given time, even although it could be the case that this agent officially had the rights to do so. In this way, learners could distinguish access rules that are actually needed, instead of leaving the establishment of such rules and modifications of policies to modelers and business practitioners alone. As shown hereafter, our contributions provide some important benefits with regard to correctness and completeness when compared with earlier techniques, allowing to improve the obtained results when executing the aforementioned analysis tasks.

2 Related Work

Process mining is a relatively young research area. Cook and Wolf [14], Agrawal et al. [15], Lyytinen and Datta et al. [16] and van der Aalst et al. [2] can be considered as fundamental works in the field.

The notion of negative events was developed in the context of the application of machine learning and classification techniques towards process discovery. Maruster et al. [17] were among the first to investigate the use of rule-induction techniques to predict dependency relationships between activities. The authors do not rely on artificial negative event generation, but apply a uni-relational classification learner on a table of direct metrics for each process activity in relation to other activities. Ferreira and Ferreira [18] apply a combination of inductive logic programming and partial-order planning techniques, where negative events are collected from users and domain experts. Similarly, Lamma et al. [19] apply an logic programming towards declarative process discovery. Unlike the approach in [18], the authors assume the presence of negative events, without providing an immediate answer to their origin. In [9], the authors extend this approach and define a "negation response" constraint in order to derive information about which activities where prohibited to occur after some other activity type. Our approach differs from these methods, since we apply a window based trace comparison technique in order to derive negative events based on a given process log, leading to a larger set of negative examples. Furthermore, instead of deriving information about prohibited state transitions in the form of declarative rules, we immediately inject the negative examples in the given traces.

Goedertier et al. [7] represent the process discovery task as a multi-relational first-order classification learning problem and use the TILDE inductive logic

programming learner for their AGNEsMiner algorithm to induce a declarative control flow model. To guide the learning process, an input event log is supplemented with artificial negative events by replaying the positive events of each process instance and by checking if a state transition of interest corresponding to a candidate negative event could occur, more specifically by investigating if other traces can be found in the event log which do allow this state transition, and present a similar history of completed activities.

3 Preliminaries

Before outlining the artificial negative event generation algorithm, some important concepts and notations that are used in the remainder of this paper are discussed.

An event log consists of events that pertain to process instances. A process instance is defined as a logical grouping of activities whose state changes are recorded as events. We thus assume that it is possible to record events such that each event refers to a task (a step in a process instance), a case (the process instance) and that events are totally ordered.

As such, let X be a set of event identifiers, P a set of case identifiers, the alphabet A a set of activity types and the alphabet E a set of event types corresponding with activity life cycle state transitions (e.g. *start*, *assign*, *restart*, *complete*, etc.). An event can then be formulated as a predicate $Event(x, p, a, e, t)$ with $x \in X$ the event identifier, $p \in P$ the case identifier and $a \in A$ the activity type, $e \in E$ the event type and $t \in \mathbb{N}$ the position of the event in its process sequence. Note that events commonly store much more information (timestamps, originators, case data, etc.), but since we do not necessarily have to deal with this additional information in the context of generating artificial negative events, this information is left out in the remainder of this paper. Furthermore, we will assume that the state transition e for each logged event equals *complete*. The function $Case \in X \cup L \mapsto P$ denotes the case identifier of an event or sequence. The function $Activity \in X \mapsto A$ denotes the activity type of an event.

Let event log L be a set of sequences. Let $\sigma \in L$ be an event sequence; $\sigma = \{x | x \in X \wedge Case(x) = Case(\sigma)\}$. The function $Position \in X \times L \mapsto \mathbb{N}_0$ denotes the position of an event in its sequence. The set X of event identifiers has a complete ordering, such that $\forall x, y \in X : x < y \vee y < x$ and $\forall x, y \in \sigma, x < y : Position(x) < Position(y)$. Let $x.y \subseteq \sigma$ be two subsequent event identifiers within a sequence σ; $x.y \Leftarrow \exists x, y \in \sigma : x < y \wedge \nexists z \in \sigma : x < z < y$. We will use this predicate in the context of single sequence which is therefore left implicit. Furthermore, a sequence of two events $x.y$ with activity types a and b respectively can be abbreviated as $\langle a, b \rangle$. Each row σ_i in the event log now represents a different execution sequence, corresponding to a particular process instance, and can be depicted as $\langle a, b, c, ..., z \rangle$ with $a, b, c, ..., z$ the activity types of the ordered events contained in the sequence.

4 Artificial Negative Event Generation Algorithm

In this section, the artificial negative event generation algorithm is discussed in more detail. Since we extend the negative event generation algorithm as introduced in AGNEsMiner, we will avoid describing some parts in excessive detail, instead referring to [7] for a more detailed overview of some steps, allowing us to focus on new contributions.

A high level overview of the algorithm can be given as follows. Negative events record that at a given position in an event sequence, a particular event cannot occur. At each position in each event trace in the log, it is examined which negative events can be recorded for this position. In a first step, frequent temporal constraints are mined from the event log. Secondly, structural information is derived from these frequent temporal constraints. Finally, negative events are induced for each grouped process instance, based on two complementing generation techniques: a window based trace comparison routine and a structural introspective technique which derives negative events from dependency information mined from an event log.

4.1 Step 1: Mining Frequent Temporal Constraints

Frequent temporal constraints are constraints that hold in a sufficient number of sequences σ within an event log L. The following list of predicates express temporal constraints that either hold or not for a particular sequence $\sigma \in L$, with $a, b, c \in A$ activity types:

$$
\begin{aligned}
Existence(1,a,\sigma) &\Leftrightarrow \exists x \in \sigma : Activity(x) = a \\
Absence(2,a,\sigma) &\Leftrightarrow \nexists x,y \in \sigma : Activity(x) = a \wedge Activity(y) = a \wedge x \neq y \\
Ordering(a,b,\sigma) &\Leftrightarrow \forall x,y \in \sigma, Activity(x) = a, Activity(y) = b : x.y \\
Precedence(a,b,\sigma) &\Leftrightarrow \forall y \in \sigma : Activity(y) = b, \exists x \in \sigma : Activity(x) = a \wedge x < y \\
Response(a,b,\sigma) &\Leftrightarrow \forall x \in \sigma : Activity(x) = a, \exists y \in \sigma : Activity(y) = b \wedge x < y \\
ChainPrec(a,b,\sigma) &\Leftrightarrow \forall y \in \sigma : Activity(y) = b, \exists x \in \sigma : Activity(x) = a \wedge x.y \\
ChainResp(a,b,\sigma) &\Leftrightarrow \forall x \in \sigma : Activity(x) = a, \exists y \in \sigma : Activity(y) = b \wedge x.y \\
ChainSeq(a,b,c,\sigma) &\Leftrightarrow (\forall z \in \sigma : Activity(z) = c, \exists x,y \in \sigma : Activity(x) = a \wedge Activity(y) = b \wedge x.y.z) \vee \\
&\quad (\forall y \in \sigma : Activity(y) = b, \exists x,z \in \sigma : Activity(x) = a \wedge Activity(z) = c \wedge x.y.z) \vee \\
&\quad (\forall x \in \sigma : Activity(x) = a, \exists y,z \in \sigma : Activity(y) = b \wedge Activity(z) = c \wedge x.y.z)
\end{aligned}
$$

For an event log L, a temporal constraint C is considered frequent if its support is greater than or equal to a predefined threshold. Let C, D be temporal constraints. The support for a temporal constraint can be defined as:

$$Supp_{\sigma \in L}(C,L) = \frac{|S|}{|L|}$$

for which S is a set containing the sequences σ for which C succeeds. Temporal constraints can also be combined to form temporal association rules of the form $C \rightarrow D$. The support of an association rule is defined as:

$$Supp_{\sigma\in L}(C\rightarrow D,L)=Supp_{\sigma\in L}(C,L)$$

The confidence of a temporal association rule is defined as:

$$Conf_{\sigma\in L}(C\rightarrow D,L)=\frac{Supp_{\sigma\in L}(C\wedge D,L)}{Supp_{\sigma\in L}(C,L)}$$

Temporal association rules are considered frequent if their support and confidence are greater than or equal to a predefined threshold. Since some activities occur more frequently than others in some event logs, the detection of frequent patterns must not be sensitive to the frequency of occurrence of a particular activity type in the event log. Consider for an example an event log L containing a large number of traces that do not contain activity type a. When checking the rule $ChainResp(a, b, \sigma)$ over all traces in L, this expression will hold true for all traces that do not contain a. Since these traces make up for a large part of the event log, the support of $ChainResp(a, b)$ would thus be high. To detect frequent patterns in an event log, it is therefore more important to look at the confidence of the association rule $Existence(1, a, \sigma) \rightarrow ChainResp(a, b, \sigma)$. The following frequent temporal association rules are derived from an event log L (left implicit):

$$\begin{aligned}
Absence(2,a) &\Leftarrow \forall a\in A: Supp_{\sigma\in L}(Absence(2,a,\sigma),L)\geq t_{absence}\\
Ordering(a,b) &\Leftarrow \forall a,b\in A: Conf_{\sigma\in L}(Existence(1,a,\sigma)\wedge Existence(1,b,\sigma)\rightarrow\\
&\qquad Ordering(a,b,\sigma),L)\geq t_{ordering}\\
Precedence(a,b) &\Leftarrow \forall a,b\in A: Conf_{\sigma\in L}(Existence(1,b,\sigma)\rightarrow\\
&\qquad Precedence(a,b,\sigma),L)\geq t_{succession}\\
Response(a,b) &\Leftarrow \forall a,b\in A: Conf_{\sigma\in L}(Existence(1,a,\sigma)\rightarrow\\
&\qquad Response(a,b,\sigma),L)\geq t_{succession}\\
ChainPrec(a,b) &\Leftarrow \forall a,b\in A: Conf_{\sigma\in L}(Existence(1,b,\sigma)\rightarrow ChainPrec(a,b,\sigma),L)\geq t_{chain}\\
ChainResp(a,b) &\Leftarrow \forall a,b\in A: Conf_{\sigma\in L}(Existence(1,a,\sigma)\rightarrow ChainResp(a,b,\sigma),L)\geq t_{chain}\\
ChainSeq(a,b,c) &\Leftarrow \forall a,b,c\in A: Conf_{\sigma\in L}(Existence(1,a,\sigma)\vee Existence(1,b,\sigma)\vee\\
&\qquad Existence(1,c,\sigma))\rightarrow ChainSeq(a,b,c,\sigma),L)\geq t_{triple}
\end{aligned}$$

Remark that our definition of the *Ordering* and *ChainSeq* temporal constraints and corresponding association rules differs from the original implementation so that the detection of these patterns is no longer sensitive to the frequency of occurrence of particular activity types. This is our first tangible improvement. We also define the following additional temporal constraint:

$$\begin{aligned}
Iteration(l,n,v,\sigma) \Leftrightarrow \exists s\in\sigma, t=Position(s,\sigma): Activity(s)=Activity(u)\wedge\forall i\in[1,l], j\in[1,n]:\\
\forall x\in\sigma, y\in v, i=Position(y,v), t+i+jl=Position(x,\sigma):\\
Activity(x)=Activity(y)\wedge l=|v|, v\subseteq\sigma, 1=Position(u,v)
\end{aligned}$$

With the corresponding association rule:

$$Iteration(l,n,v) \Leftarrow \forall v \subseteq \sigma, \sigma \in L, l=|v|, n=2, 0<l\leq 3: Conf_{\sigma \in L}(\forall x \in v: Existence(1, Activity(x), \sigma) \rightarrow Iteration(l,n,v,\sigma)) \geq t_{iteration}$$

The *Iteration* temporal constraint holds in a sequence σ if another sequence v with length l iterates n times in σ. In order to keep the number of computations under control, we limit the discovery of iterations to length 3 and less, which repeat for minimally 2 times; $n = 2$ and $0 < l \leq 3$.

The derivation of temporal frequent constraints can be compared with the more general problem of mining event sequences and episodes in an event log using apriori-like techniques, see for instance [20, 21, 22]. An alternative but ultimately similar method to mine structural behavior from an event log is suggested by Maggi et al. [23], who apply temporal logic property verification techniques, formalized in the Declare modeling language to discover structural behavior.

4.2 Step 2: Deriving Structural Information: Parallelism, Locality and Recurrence

Now that a set of temporal frequent constraints is constructed, it becomes possible to derive information about parallelism and locality of pairs of activities. Based on the constraints above, two derivation rules: $Parallel(a, b)$ (symmetric) and $Local(a, b)$ (non-symmetric), can be defined. The former denotes that two activities $a, b \in A$ can occur concurrently, while the latter states that two activities occur in a serial manner, denoting an explicit dependency. For a more detailed description of these derivation rules, see [7].

Additionally, the *Iteration* temporal constraint as defined in the previous section allows us to formulate a loop discovery heuristic. However, loops cannot immediately be derived from the set of *Iteration* temporal constraints. To see why this is the case, consider the sequence $\langle a, b, c, d, b, c, d, b, c, d, e\rangle$. The following Iteration constraints hold in this sequence: $Iteration(3, 2, \langle b, c, d\rangle)$, $Iteration(3, 2, \langle c, d, b\rangle)$ and $Iteration(3, 2, \langle d, b, c\rangle)$. Since we are only concerned with the loop following from the first Iteration constraint, with the start activity in the correct, first position, we define $Loop(v)$ as such:

$$\forall v \subseteq \sigma, \sigma \in L, 1=Position(u,v), l=|v|: Iteration(l,2,v) \wedge \exists a \in A: Local(a,u) \Rightarrow Loop(v)$$

4.3 Step 3: Generating Artificial Negative Events

Window Based Negative Event Generation. Once parallelism, recurrence and locality information is derived, negative examples can be introduced in given process instances. Negative events record that at a particular position in an event sequence, a particular event cannot occur. At each position k in each event sequence τ_i, it is examined which negative events can be introduced at this

position. In a first step, the event log is made more compact by grouping process instances that have identical sequences $\sigma \in L$ into grouped process instances $\tau \in M$. In the next step, all negative events are induced for each grouped process instance τ_i (the "positive trace" under consideration) by checking at any given positive event $x_k \in \tau_i$ whether another event of interest (a "candidate negative event") z_k with activity type $b \in A \setminus \{Activity(x_k)\}$ also could occur. Thus, for each positive event $x_k \in \tau_i$, it is tested whether there exists a sequence $\tau_j \neq \tau_i \in M$ in the event log in which an event y_k has taken place that is similar to z_k and where both sequences present a similar history up until that point. Note that we consider two events being similar if their activity types are identical. If such a similar "disproving sequence" can not be found, such behavior is not present in the event log L, meaning that the candidate event indeed cannot occur. Consequently, candidate negative event z_k cannot be disproved and is added at position k in the positive sequence τ_i. Finally, the negative events in the grouped process instances are used to induce negative events into the similar non-grouped sequences. Usually, a large number of negative events can be generated, so that a probability π is introduced as a threshold for injecting negative events into the ungrouped sequences.

To address the problematic nature of the completeness assumption of an event log under recurrent (loops) and concurrent (parallelism) behavior, we exploit the previously mined parallelism and loop information to generate parallel and looping variants of traces, by swapping parallel activity types and inserting or removing loops where possible. We now thus compare the positive trace under consideration τ_i with each $\tau_j^{\sim} \in AllVariants(\tau_j)$, with $AllVariants$ a function which returns the set of all parallel and loop variants of sequence τ_j. Generating these variants results in a larger number of traces which will be used when evaluating a negative event, thus resulting in a weaker completeness assumption. The addition of the generation of variants based on recurrence is a second contribution.

Even when parallel and loop variants are considered, incorrect negative events could still be induced, due to the possible recursive and complex nature of recurrence and concurrency (nested loops, for example). Instead of trying to deal with this complex behavior by adjusting the *Loop* and *Iteration* constraints further, a window size parameter $windowSize$ is introduced to limit the number of events which are compared when evaluating a candidate negative event (i.e. how far we look back before $x_k \in \tau_i$ and $y_k \in \tau_j^{\sim}$). The larger the window size, the less probable that a similar subsequence is contained by the other sequences in the event log, and the more probably that a candidate negative event will be introduced. Reducing the window size makes the completeness assumption less strict. An unlimited window size (a maximum-length comparison between sequences) results in the most strict completeness assumption. Note that history-dependent processes generally will require a larger window size to correctly detect all non-local dependencies. When the window size is limited to 1, it is no longer possible to take into account non-local dependencies, so that negative events following from this behavior will not be generated. This underlines the trade-off as discussed

before in the introduction. Using a higher window size leads to the generation of more valuable negative events, that is, negative events derived from non-local dependencies. On the other hand, using an increased window also leads to a more strict completeness assumption, so that candidate negative events are less likely to be disproved, leading to the possible introduction of incorrect negative examples.

Contrary to the strict manner of comparing windows found in the original definition of the window comparison routine found in [7] – which stated that the position of the completed event under consideration (x_k) in its trace (τ_i) must be equal to the position of the event with activity type equal to the activity type of the proposed negative event (y_k) in the candidate disproving trace ($\tau_j^{\sim}$), we compare the window τ_i of the original "positive" trace with *each* possible window in the "candidate disproving" trace $\tau_j^{\sim}$, meaning each sequence of events before an event with activity type equal to the activity type of the current candidate negative event under consideration. An example can clarify this principle. Consider the positive trace $\tau_i = \langle a, b, c, d, f \rangle$ and $\tau_j^{\sim} = \langle a, b, c, b, c, e, f \rangle$ a candidate disproving trace under consideration. Assume that we are currently checking to see if candidate activity e could also occur instead of d in τ_i. In cases where a strict window is used, with size equal to, say, 2, the candidate disproving trace $\tau_j^{\sim}$ fails to disprove the negative event, since $Position(d, \tau_i) \neq Position(e, \tau_j^{\sim})$. Instead of doing so, we now compare the window $\langle b, c \rangle$ in τ_i with each possible window in $\tau_j^{\sim}$. In this case, the window of size 2 before activity e in $\tau_j^{\sim}$ is also equal to $\langle b, c \rangle$. This leads to a correct rejection of the candidate negative event. The full artificial negative event generation algorithm using this "dynamic" window is listed in Algorithm 1.

Note that, depending on the window size configured, cases might now exist where the size of the window in the original trace τ_i is unequal to the size of a window in a candidate disproving trace $\tau_j^{\sim}$. In cases where the window in $\tau_j^{\sim}$ is larger than the window in τ_i, an effective window with size equal to the window used in τ_i is used. When the window in $\tau_j^{\sim}$ is smaller than the window in τ_i, using the smallest window could potentially lead to the rejection of negative events, even when a high window size parameter was set. An example can help to illustrate this. Consider again the positive trace $\tau_i = \langle a, b, c, d, f \rangle$ and $\tau_j^{\sim} = \langle a, b, c, b, c, e, f \rangle$ the candidate disproving trace under consideration. Assume now we are currently checking to see if candidate activity b could also occur instead of d in τ_i. Based on this information, two windows can be defined in $\tau_j^{\sim}$ which can be used to check the validity of the candidate negative event at hand: $\langle a \rangle$ and $\langle a, b, c \rangle$. For the latter, no problem exists, as this window is as long as the window $\langle a, b, c \rangle$ in τ_i. On the other hand, the other window ($\langle a \rangle$) is smaller than the window in τ_i, and thus the similarity check between these two windows might differ from the actual window size parameter which was configured by the user. Although this does not lead to the generation of incorrect negative events (or any additional events, in fact), we define a parameter $minimumWindowSize$ to denote a minimum required length for a window to be used in the negative

Algorithm 1. Artificial event generation algorithm with a dynamic window

-- group similar sequences $\sigma \in L$ into $\tau \in M$
$M := GroupLog(L)$
-- generate artificial events
$N := \emptyset$
for each $\tau_i \in M$ do
  for each $x_k \in \tau_i$ do
    $k := Position(x_k, \tau_i)$
    for each $b \in A \setminus \{Activity(x_k)\}$ do
      if $\nexists \tau_j^{\sim} \in AllVariants(\tau_j) : \forall \tau_j \in M \wedge \tau_j^{\sim} \neq \tau_i \wedge$
        $\forall y_k \in \tau_j^{\sim}, j = Position(y_k, \tau_j^{\sim}), Activity(y_k) = b :$
          $\forall y_l \in \tau_j^{\sim}, j - l = Position(y_l, \tau_j^{\sim}), k - l = Position(x_l, \tau_i),$
            $0 < l \leq windowSize : Activity(y_l) = Activity(x_l)$
      then $z_k := Event(x, Case(\tau_i), b, completeRejected, k)$
        -- with x a new event identifier
        $N := N \cup \{NegativeEvent(z_k, k, \tau_i)\}$
-- induce negative events in non grouped sequences $\sigma \in L$ with injection frequency π
$L_n := InduceEvents(N, L, \pi)$

event rejection procedure. Setting this parameter to -1 denotes that the window in $\tau_j^{\sim}$ should be at least of the same size as the current window in τ_i.

Using this dynamic window extension now allows us to drastically weaken the completeness assumption made by the artificial negative event generation process. Using a dynamic window with size 1 indeed assumes only that each binary sequence of two activities is somewhere present in the given event log, or can be generated from the given event log using parallel and loop variant calculation as described above. This weakens the completeness assumption equal to the one made by the formal alpha-miner learner [2].

Finally, remark the absence of a "forward window"; we only consider the history of completed events when investigating which activities could not be completed at a certain point in the process instance. The reason for this is straightforward: at the time of investigating a current state transition, the future of the process instance at hand is still unknown. Therefore, only historical facts can be considered in the negative event generation process.

Dependency Based Negative Event Generation. As a fourth and final extension, instead of using a window based trace comparison algorithm, we present an alternative way of generating artificial negative events, based on discovered structural information as given by the temporal frequent constraints and association rules. Both explicit dependency (locality), and long-distance dependency information can be applied towards the induction of negative events.

Explicit Dependencies (Locality). It is possible to generate negative events based on locality information (i.e. explicit dependencies). As a rule of thumb, negative events with a certain activity type can be added before a given completed event in a process instance at position k when this activity type is not locally dependent on the activity type of the event completed at the previous position $(k-1)$[1]:

$$\forall \tau \in M, 1 \leq k \leq |\tau|, k = Position(x_k, \tau), a \in A, a \neq Activity(x_k), k-1 = Position(x_{k-1}, \tau):$$
$$\sim Local(Activity(x_{k-1}), a) \Rightarrow Event(x, Case(x_k), a, completeRejected, k)$$

Note that, when $k = 1$, x_{k-1} will be set to a dummy, non-existing event, so that constraints involving this dummy event (e.g. $Local(x_0, x_1)$) always evaluates as being false. This constraint is omitted from the above and following definitions for reasons of clarity.

Long-Distance Dependencies, Implicit Dependencies. We define the following structural derivation rules to mine all dependencies between activity types (both implicit and explicit), similar to *Local* (explicit dependencies only):

$$\forall a, b \in A: (Precedence(a,b) \vee Response(a,b)) \wedge \sim Parallel(a,b) \Rightarrow Dependence(a,b)$$
$$\forall a, b \in A: (Precedence(a,b) \wedge Response(a,b)) \wedge \sim Parallel(a,b) \Rightarrow StrongDependence(a,b)$$

Based on these dependencies, the rule of thumb defined above can be expanded. Even negative events with a certain activity type that is locally dependent on the activity type of the event completed at the previous position $(k-1)$ can be added before a given completed event (position k), so long as not *all* activities on which the negative event under consideration is *strongly* dependent (*StrongDependence*) on were completed before, or so long as *no single* activity on which the negative event under consideration is dependent on (*Dependence*) has completed before:

$$\forall \tau \in M, 1 \leq k \leq |\tau|, k = Position(x_k, \tau), a \in A, a \neq Activity(x_k), k-1 = Position(x_{k-1}, \sigma),$$
$$\upsilon = \{Activity(i) | i \in \sigma, Position(i, \sigma) < k\}: \exists b \in A, b \notin \upsilon:$$
$$StrongDependence(b,a) \Rightarrow Event(x, Case(x_k), a, completeRejected, k)$$

and:

$$\forall \tau \in M, 1 \leq k \leq |\tau|, k = Position(x_k, \tau), a \in A, a \neq Activity(x_k), k-1 = Position(x_{k-1}, \sigma),$$
$$\upsilon = \{Activity(i) | i \in \sigma, Position(i, \sigma) < k\}: \nexists b \in A, b \in \upsilon:$$
$$Dependence(b,a) \Rightarrow Event(x, Case(x_k), a, completeRejected, k)$$

[1] Remark that we use negation-as-failure ($\sim$) rather than normal logical negation ($\neg$) to denote that the absence of a frequent temporal constraint is derived from the absence of sequences in the event log that portray this behavior. In this context, the reader may ignore the exact semantic differences between the two notations, as $\sim p \Rightarrow \neg p$ under a closed-world assumption.

Finally, we can also use the *StrongDependence* construct to suggest an optimal minimum window size, i.e. a window size which is able to capture pairs of activity types between which an implicit (long-distance) dependency relation exists. To do so, we need to restrict *StrongDependence* a bit further to drop strong dependencies which do not correspond with an implicit dependency in the underlying process model. For example, a *StrongDependence* construct can always be found between starting and ending activities. However, a starting activity is always followed by the ending activity, so that this dependency is not an implicit one. Deriving a window size from this construct would lead to a useless suggestion, as this would give the same result as when using an unlimited window. Therefore, we restrict our search to unique strong dependencies (derived with the *UniqueStrongDependence* rule below) where activity types are strongly dependent on one other activity type only, which is a good indication for the presence of an implicit dependency.

$$\forall a,b \in A: StrongDependence(a,b) \wedge \nexists c \in A, c \neq a: StrongDependence(c,b) \Rightarrow UniqueStrongDependence(a,b)$$

$$MinimalWindowSize(M) = Max_{a,b}(Min_{x,y,\tau}(Position(x,\tau) - Position(y,\tau)))$$
$$\text{with: } a,b \in A, UniqueLongDistanceDependence(a,b),$$
$$\tau \in M, x \in \tau, y \in \tau, Activity(x) = a, Activity(y) = b, Position(x,\tau) < Position(y,\tau)$$

Note that, when sequences are present in the event log which contain multiple events corresponding to the same activity type (e.g. for process models containing loops), $Position(x,\tau) - Position(y,\tau)$ is computed such that the resulting difference between the two events is minimal and greater than zero.

5 Experimental Results and Discussion

We have implemented our revised artificial negative event generation technique in ProM6 [24]. We test the improvements above with the "DriversLicenseLoop" log, an artificial process log which has been used before by de Medeiros et al. [3] to evaluate the GeneticMiner discovery algorithm. The drivers license process is interesting, since it contains parallelism, recurrence, duplicate tasks and implicit dependencies. To compare the different settings of the artificial event generation algorithm, a process log containing 350 process instances was used (87 distinct traces, 11 activity types).

Table 1 lists the various parameter configurations used the evaluate the artificial event generation technique. For each of the configurations, all generated negative events were introduced in the given event log. The "Window Generation" parameter denotes the use of window based artificial negative event generation, "Window Size" denotes the size of the window, "Dynamic Window" denotes the use of the dynamic window improvement with a minimum

required window size "Minimum Window Size". "Structural Generation" defines if dependency based artificial negative event generation is performed, either on its own ("strucOnly"), or together with window based negative event generation ("strucNonDynamicWs-1" and following are obtained by merging a window based generated set of negative events with the set of negative events obtained from "strucOnly"). We use window sizes -1 (unlimited), 3 (suggested by $MinimalWindowSize(M)$) and 1 (most limited) to test the event generation procedure. When a dynamic window is used, we use both -1 (candidate disprove window must be as long as window in positive trace) and 1 (no effective minimum) as required minimum window values. "Original" configuration identifiers correspond with a parameter setting which could be obtained with the original version (i.e., no improvements) of the artificial event generation algorithm.

Table 1. Used parameter setting configurations for the artificial event generation tests

Parameter Configuration Identifier	Window Generation	Window Size	Dynamic Window	Minimum Window Size	Structural Generation
originalWs-1	yes	-1	no	–	no
originalWs3	yes	3	no	–	no
originalWs1	yes	1	no	–	no
dynamicWs-1MinWs-1	yes	-1	yes	-1	no
dynamicWs3MinWs-1	yes	3	yes	-1	no
dynamicWs1MinWs-1	yes	1	yes	-1	no
dynamicWs-1MinWs1	yes	-1	yes	1	no
dynamicWs3MinWs1	yes	3	yes	1	no
dynamicWs1MinWs1	yes	1	yes	1	no
strucOnly	no	–	–	–	yes
strucNonDynamicWs-1	yes	-1	no	–	yes
strucNonDynamicWs3	yes	3	no	–	yes
strucNonDynamicWs1	yes	1	no	–	yes
strucDynamicWs-1MinWs-1	yes	-1	yes	-1	yes
strucDynamicWs3MinWs-1	yes	3	yes	-1	yes
strucDynamicWs1MinWs-1	yes	1	yes	-1	yes
strucDynamicWs-1MinWs1	yes	-1	yes	1	yes
strucDynamicWs3MinWs1	yes	3	yes	1	yes
strucDynamicWs1MinWs1	yes	1	yes	1	yes

Table 2 gives the results for each of the above defined parameter setting configurations. We compare the results for the various parameter configurations with two given sets of negative events. A "naive generation method" constructs a set of negative events by injecting at each position in a trace a negative event for each activity type, except the activity type equal to the (completed) event at the current position. Note that even when this naive method is used, the number of incorrect negative events in respect to the total negative events is rather low. This is an indication towards the fact that the given event log gives a good coverage of all possible execution traces as allowed by the underlying process model. The "Fully Correct Log" was constructed based on the given Petri net used to simulate the drivers license process. Of course, in real life cases, such a reference model is

Table 2. Results of the DriversLicenseLoop experiment under various configurations

Parameter Configuration Identifier	Incorrect Negative Events	Total Negative Events	Correctness	Completeness
Fully Correct Log	0	44866	100%	100%
Naive Generation Method	2484	47350	0%	100%
originalWs-1	642	45508	74,2%	100%
originalWs3	621	45487	75,0%	100%
originalWs1	621	44866	75,0%	98,6%
dynamicWs-1MinWs-1	642	45508	74,2%	100%
dynamicWs3MinWs-1	0	44866	**100%**	**100%**
dynamicWs1MinWs-1	0	42382	100%	94,5%
dynamicWs-1MinWs1	642	45508	74,2%	100%
dynamicWs3MinWs1	0	44866	**100%**	**100%**
dynamicWs1MinWs1	0	42382	100%	94,5%
strucOnly	0	40469	100%	90,2%
strucNonDynamicWs-1	642	45508	74,2%	100%
strucNonDynamicWs3	621	45487	75,0%	100%
strucNonDynamicWs1	621	45349	75,0%	99,7%
strucDynamicWs-1MinWs-1	642	45508	75,2%	100%
strucDynamicWs3MinWs-1	0	44866	**100%**	**100%**
strucDynamicWs1MinWs-1	0	43969	100%	98,0%
strucDynamicWs-1MinWs1	642	45508	74,2%	100%
strucDynamicWs3MinWs1	0	44866	**100%**	**100%**
strucDynamicWs1MinWs1	0	43969	100%	98,0%

unavailable, preventing the construction of a fully correct set of negative events by which the artificial induction results can be evaluated. For each parameter configuration, we calculate the correctness and completeness ratio. Correctness is defined as one minus the ratio of incorrect negative events to the number of incorrect negative events generated by the naive method. Completeness is defined as the ratio of correct negative events over the full number of possible, correct negative events, as given by the fully correct log.

The following conclusions can be derived from the results. First, the inherent trade-off between correctness and completeness becomes apparent here, as most configurations show an inverse relation between the two requirements. Second, we note that no single window size configuration is able to generate a set of negative events which is both correct and complete when using the original version of the artificial event generation algorithm. Next, using a strict window size (1) in combination with the dynamic window improvement leads to a set of negative events which is fully correct, albeit not complete. Constructing a set of negative events which is both complete and correct is possible if the window size is increased to 3 (suggested by investigating the structure of implicit dependencies – denoted in bold case in Table 2), or by using non-window dependency based generation, which also leads to an acceptable completeness value (98%). Moreover, using dependency-based generation ensures the addition of "non-trivial" negative events, derived from implicit dependencies, which proves especially helpful in a later phase when the set of negative events is used for evaluation or discovery tasks. The results also deal with another concern: even although we have defined a large number of parameters, two straightforward, well performing defaults can be suggested: either apply window based generation with

a dynamic window of size 1 in conjunction with dependency based event generation, or only apply window based generation with a window size equal to the suggested window size.

6 Conclusions and Future Work

In this paper, we propose an improved artificial negative event generation method, building upon [7] in order to derive sets of negative events which are both correct and complete. The construction of event logs with artificial negative events can be expected to be valuable in multiple settings: first, supervised learners can now be deployed in order to perform a process discovery task. Second, artificial induction of negative events can be applied towards evaluation purposes as well, where an event log, supplemented with negative events, is used to assess fitness and precision of process discovery algorithms. Finally, since negative events capture which actions in a process could not occur, compliance and conformance related analysis tasks present themselves as a natural area of application for negative events, although it should be noted that, in order to perform this last set of tasks, the artificial negative event generation technique as described here should be extended to take state transitions other than completed events into account, together with event originator (agent) and case data in order to fully utilize all available information. In future work, we aim at validating our novel approach, by extending the set of artificial logs included in the experiment and by examining the different parameter configurations. In this way, we aim at making event logs with artificially generated negative events widely available.

Acknowledgements. We would like to 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.

References

[1] van der Aalst, W., Reijers, H., Weijters, A., van Dongen, B., Alves de Medeiros, A., Song, M., Verbeek, H.: Business process mining: An industrial application. Information Systems 32(5), 713–732 (2007)
[2] van der Aalst, W., Weijters, A.J.M.M., Maruster, L.: Workflow mining: discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1128–1142 (2004)
[3] de Medeiros, A., Weijters, A., van der Aalst, W.: Genetic process mining: an experimental evaluation. Data Mining and Knowledge Discovery 14(2), 245–304 (2007)
[4] de Medeiros, A., van Dongen, B., van der Aalst, W.: Process mining: Extending the alpha-algorithm to Mine Short Loops (2004)
[5] Weijters, A., van der Aalst, W., de Medeiros, A.: Process mining with the heuristics miner-algorithm (2006)
[6] Wen, L., van der Aalst, W., Wang, J., Sun, J.: Mining process models with non-free-choice constructs. Data Mining and Knowledge Discovery 15(2), 145–180 (2007)

[7] Goedertier, S., Martens, D., Vanthienen, J., Baesens, B.: Robust Process Discovery with Artificial Negative Events. Journal of Machine Learning Research 10, 1305–1340 (2009)
[8] Blockeel, H., De Raedt, L.: Top-down induction of first-order logical decision trees. Artificial Intelligence 101(1-2), 285–297 (1998)
[9] Chesani, F., Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Exploiting Inductive Logic Programming Techniques for Declarative Process Mining. In: Jensen, K., van der Aalst, W.M.P. (eds.) Transactions on Petri Nets and Other Models of Concurrency II. LNCS, vol. 5460, pp. 278–295. Springer, Heidelberg (2009)
[10] Muggleton, S.: Inductive logic programming. In: Proceedings of the 1st International Conference on Algorithmic Learning Theory, pp. 42–62 (1990)
[11] 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 (LNAI), vol. 4894, pp. 132–146. Springer, Heidelberg (2008)
[12] Dzeroski, S., Lavrac, N.: Inductive Logic Programming: Techniques and Applications. Ellis Horwood, New York (1994)
[13] De Weerdt, J., De Backer, M., Vanthienen, J., Baesens, B.: A robust f-measure for evaluating discovered process models. In: IEEE Symposium Series in Computational Intelligence (2011)
[14] Cook, J., Wolf, A.: Discovering models of software processes from event-based data. ACM Transactions on Software Engineering and Methodology 7(3), 215–249 (1998)
[15] 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. 467–483. Springer, Heidelberg (1998)
[16] Lyytinen, K., Mathiassen, L., Ropponen, J., Datta, A.: Automating the discovery of as-is business process models: Probabilistic and algorithmic approaches. Information Systems Research 9(3), 275–301 (1998)
[17] Maruster, L., Weijters, A., van der Aalst, W., van den Bosch, A.: A Rule-Based Approach for Process Discovery: Dealing with Noise and Imbalance in Process Logs. Data Mining and Knowledge Discovery 13(1), 67–87 (2006)
[18] Ferreira, H., Ferreira, D.: An integrated life cycle for workflow management based on learning and planning. International Journal of Cooperative Information Systems 15(4), 485–505 (2006)
[19] 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)
[20] Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules. VLDB 1215, 487–499 (1994)
[21] Huang, K., Chang, C.: Efficient mining of frequent episodes from complex sequences. Information Systems 33(1), 96–114 (2008)
[22] Mannila, H., Toivonen, H., Inkeri Verkamo, A.: Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery 1(3), 259–289 (1997)
[23] Maggi, F., Mooij, A., van der Aalst, W.: User-guided discovery of declarative process models. In: IEEE Symposium on Computational Intelligence and Data Mining (2011)
[24] van der Aalst, W., van Dongen, B., Rozinat, A., Günther, C., Verbeek, E.: Prom: The process mining toolkit. In: de Medeiros, A.K.A., Weber, B. (eds.) BPM (Demos). CEUR Workshop Proceedings, vol. 489, CEUR-WS.org (2009)