An improved process event log artificial negative event generator

Seppe vanden Broucke, Jochen De Weerdt, Jan Vanthienen, and Bart Baesens
An Improved Process Event Log Artificial Negative Event Generator

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

Abstract—Process mining is the research area that is concerned with knowledge discovery from event logs and is often situated at the intersection of the fields of data mining and business process management. Although the term entails a collection of a-posteriori analysis methods for extracting knowledge from event logs, most of the attention in the process mining literature has been given to process discovery techniques, focusing specifically on the extraction of control-flow models from event logs. Process discovery (and process mining in general) faces notable difficulties. One difficulty 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 artificially generate negative events, based on a technique first formulated in the context of the AGNEsMiner process discovery algorithm. In the original version of the algorithm, a configurable completeness assumption is defined by proposing a window size parameter and a negative event injection probability. We present several improvements to make this completeness assumption less strict, in order to prevent the introduction of falsely induced negative events in cases where an event log does not or can not capture all possible behavior, while still allowing for a complete set of negative events, including those which are derived from non-local, history-dependent behavior.

Index Terms—process mining, process discovery, event logs, negative events.

1 INTRODUCTION

1.1 Process Mining

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. Information support systems such as Enterprise Resource planners (ERP) and Workflow Management systems (WiMS) often log an abundance of data about the way people conduct day-to-day practices and thus conceal an unexploited reservoir of knowledge, capturing what business activities happened at certain moments in time. The vast quantity of available events, however, makes it difficult to analyze these event logs using descriptive statistics only. The research area that is concerned with knowledge discovery from event logs is called process mining [1, 2] and is often situated at the intersection of the fields of data mining and business process management (BPM). Process mining is related to data mining because of the shared goal of learning from large data repositories [3]. Similarly, process mining can be associated with the field of BPM as one of its major objectives is gaining insight into business operations. As a result, process mining fits within the diagnosis phase of the BPM life cycle [4].

Although the term process mining entails a collection of a-posteriori analysis methods for extracting knowledge from event logs, most of the attention in the process mining literature has been given to process discovery techniques [2, 5], focusing specifically on the extraction of control-flow models from event logs. Process discovery (and process mining in general), however, faces some particular difficulties. One difficulty 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.

A large number of algorithms have been developed to discover control-flow models from event logs that allow to examine the real execution traces of business activities [2, 5, 6, 7, 8, 9]. Process discovery (and process mining in general), however, faces some particular difficulties. One difficulty 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 and consequently cannot guide the discovery task [10]. In addition, process models frequently display complex structural behavior, which makes the hypothesis space of process mining algorithms harder to navigate. The main problematic constructs are: non-free choice (non-local, history-dependent behavior), invisible activities and duplicate activities [5]. 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. Note that event logs sometimes do contain negative events in a natural manner. Access and security logs, for example, contain detailed information about users that were refused authorization to perform a particular task, or to access a particular resource. In many cases, however, information systems do not reveal or capture these negative events.

1.2 Generating Negative Events in Incomplete Event Logs

Several methods have been proposed in literature to tackle the issue of negative events in process event logs, often in the context of supervised classification techniques and the application of machine learning on process discovery, the latter of which requires the presence of negative examples in order to distinguish the right hypothesis from an infinite number of possible grammars that fit the positive examples [11]. In the next section a detailed overview of related literature is given.

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 tradeoff between these two goals is due to the completeness assumption made over an event log when generating artificial negative events. Under its most strict form, a completeness assumption requires 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.

Assuming a strict completeness assumption as defined above is, however, often unrealistic in practice. First, when loops can occur in the process model at hand, the number of possible traces can be infinite, so that no single bounded event log can possibly contain all trace variants. Second, when parallelism is present, an exponential number of execution paths can be defined, corresponding with the order in which the concurrent task are handled. Finally, even without loops or parallelism but just N binary choices, the number of possible traces can still be \(2^N\) [2]. Therefore, many process discovery algorithms make a weaker completeness assumption. For example, the formal \(\alpha\)-algorithm [2] only derives information from explicit dependencies, meaning that an event log that contains each possible sequence of activities (one directly following the other) in one of its

traces suffices. Remark that this does not entail that the \(\alpha\)-algorithm will be able to correctly mine each model using just this information, but only that this is the initial assumption made over the given input.

In this paper, we propose an artificial negative event generation method based on a technique first introduced in the AGNEsMiner process discovery algorithm [10]. In the original version of the algorithm, a configurable completeness assumption is defined by proposing a window size parameter and a negative event injection probability. We present several improvements in order to make this completeness assumption more configurable and the generation procedure more robust so that the introduction of falsely induced 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). Ensuring a correct induction of negative events proves especially helpful in practice when these events are subsequently used for evaluation purposes. Goedertier et al. [10] define two evaluation metrics to assess recall and specificity (i.e. the amount of behavior present in the event log which is captured by the proposed process model) using negative events. De Weerdt et al. [12] additionally define a precision measure based on negative events to gauge whether a mined process model does not underfit the behavior present in the event log.

To summarize, the goal of the improvements presented hereafter is inherently related to the correctness-completeness tradeoff as described above. First, we always want to allow for a parameter configuration so that the completeness assumption made by the event generation algorithm is as weak as or weaker than the completeness assumption made by the discovery algorithm(s) used in a subsequent step (with the goal of either discovery or evaluation), so that no incorrect artificial negative events are introduced which could throw off discovery algorithms using these negative examples or skew evaluation results (correctness). Second, we want to generate all correct negative events, especially those following from complex behavior, so that these artificial negative events provide valuable information for subsequent discovery algorithms or allow to distinguish performance differences between discovery algorithms in an evaluation context. Although these two goals counteract each other (due to the completeness assumption made), we show that the proposed improvements help to improve both objectives in a simultaneous manner.

2 Related Work

Initial approaches to process discovery were investigated by Cook and Wolf [13], who apply three different methods in the context of software engineering processes to approach the discovery of models from a statistical, algorithmic and probabilistic perspective. The idea of
applying process discovery specifically towards workflow management systems stems from Agrawal et al. [14] and Lytyinen and Datta et al. [15]. The α-algorithm can be considered to be a theoretical learner, for which van der Aalst et al. [2] prove that is is able to learn an important class of workflow nets from complete event logs with respect to all allowable binary sequences and the absence of noise. This learner underlined the value of process discovery in a general setting of process mining, together with the development of the ProM framework [16], which now contains a large amount of plugins, implementing an abundant amount of other process discovery techniques, such as the $\alpha^+$-algorithm [9], HeuristicsMiner [8] and GeneticsMiner [5].

Authors have proposed to use machine learning and classification techniques in the context of process discovery as well, wherein the notion of negative events was developed. Maruster et al. [17] were among the first to investigate the use of rule-induction techniques to predict dependency relationships between activities. The authors use the uni-relational classification learner RIPPER [18] on a table of direct metrics for each process activity in relation to other activities, which is generated in a pre-processing step. This conversion is performed in order to deal with the absence of negative events, which are thus not considered in the discovery task.

Ferreira and Ferreira [19] apply a combination of inductive logic programming and partial-order planning techniques to process mining. In this context, negative events are collected from users and domain experts who indicate whether a proposed execution plan is feasible or not, iteratively combining planning and learning to discover a process model.

Lamma et al. [20, 21, 22] apply an extension of logic programming, SCIFF, towards declarative process discovery. Process discovery techniques using SCIFF also requires the presence of negative events (or “non-compliant” traces), but unlike the approach of Ferreira and Ferreira, the authors assume the presence of negative events, without providing an immediate answer to their origin.

Goedertier et al. [10] 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 the discriminating preconditions that determine whether an event can take place or not, given a history of events of other activities. These preconditions are then converted to a graphical model after applying a pruning and post-processing step. To guide the learning process, an input event log is supplemented with induced 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. If no such similar trace can be found, a negative event is induced. This process of artificially generating negative events is described in a more formal way by Algorithm 1.

Supplementing event logs with artificial negative events makes the completeness assumption that all possible trajectories in the process model to be learned have corresponding process instances in the event log. Otherwise, a large number of negative events could be falsely induced, due to the fact that there are not enough similar traces present in the event log in order to prove the possible occurrence of a considered state transition and consequently disprove the validity of a candidate negative event. A large number of process instances is thus required in order to correctly induce negative events. As mentioned in the introduction, processes containing concurrent (parallelism) and recurrent (loops) behavior exponentially increase the number of allowed traces which can occur, so that the completeness assumption can become problematic. To address the latter problem, AGNESMiner implements a window size parameter to restrict the number of events which are compared when evaluating a candidate negative event. The problem of concurrent behavior is addressed by generating parallel variants for each used trace, by using structural information induced from the event log based on frequent temporal constraints. Mannila and other scholars [23, 24, 25] describe the more general problem of mining event sequences and episodes in an event log, using apriori-like techniques, which is applied here to mine the temporal frequent constraints from which structural information is derived. An alternative method to mine structural behavior from an event log is suggested by Maggi et al. [26], who apply temporal logic property verification techniques, formalized in the DECLARE modeling language to discover structural behavior.

Artificial induction of negative events is not only useful in a process discovery context, but can also be applied towards evaluation purposes, where an event log, supplemented with negative events, is used to assess accuracy and precision of process discovery algorithms. The behavioral recall ($r_B$), behavioral specificity ($s_B$) and behavioral precision ($p_B$) [12, 10] metrics all make use of event logs supplemented with negative events to compose a confusion matrix in concordance with standard metric definitions in the field of data mining.

**Algorithm 1 Artificial negative event generation in an event log.**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>foreach process instance $t_i \in$ event log $L$ do</td>
</tr>
<tr>
<td>2</td>
<td>foreach event $e(\cdot,k) \in t_i$ at position $k$ do</td>
</tr>
<tr>
<td>3</td>
<td>foreach activity state transition $a$ of interest do</td>
</tr>
<tr>
<td>4</td>
<td>if $\exists t_j \in L : t_i \neq t_j \wedge (\forall l,k :</td>
</tr>
<tr>
<td>5</td>
<td>similar($e(i,j),e(i,k)) \wedge similar(a,e(j,k))$)</td>
</tr>
<tr>
<td>6</td>
<td>then record negative event with state transition $a$ at position $k$ in $t_i$</td>
</tr>
</tbody>
</table>
3 Preliminaries

Before continuing with the introduction of the artificial negative event generation algorithm improvements, some important concepts and notations that are used in the remainder of this paper are given in this section.

3.1 Event logs

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. In the remainder of this paper, we will use the following notations.

Let \( X \) be a set of event identifiers, \( P \) 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 \( \text{Event}(x,p,a,e,t) \) 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 we do not have to deal with this additional information in the context of generating artificial negative events. Since we are primarily interested in using the generated artificial events in a process discovery context (i.e. a control-flow perspective), we furthermore assume that the state transition \( e \) for each logged event equals complete. Figure 1 shows an example event log and its corresponding list of predicates. The function \( \text{Case} \in X \cup L \mapsto P \) denotes the case identifier of an event or sequence. The function \( \text{Activity} \in X \mapsto A \) denotes the activity type of an event.

Let event log \( L \) be a set of sequences. Let \( \sigma \in L \) be an event sequence; \( \sigma = \{x| x \in X \land \text{Case}(x) = \text{Case}(\sigma)\} \). The function \( \text{Position} \in X \times L \mapsto \mathbb{N}_0 \) denotes the position of an event in its sequence. The set \( X \) of event identifiers has a complete ordering, such that \( \forall x, y \in X : x < y \lor y < x \) and \( \forall x, y \in \sigma, x < y : \text{Position}(x) < \text{Position}(y) \). Let \( x, y \subseteq \sigma \) be two subsequent event identifiers within a sequence \( \sigma \). We will use this predicate in a 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 \( \sigma_i \) 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.

3.2 Petri nets

Figure 2 shows a Petri net representation of an ordering process, containing loops, choice and parallelism. Petri nets provide a graphical, formal language to represent, analyze, verify and simulate dynamic behavior [27]. A Petri net consists of places, tokens and arcs. Places are drawn as circles, can contain tokens and are an indication of state. Each different distribution of tokens over the Petri net places indicate a different state, which is called a marking. Transitions are drawn as rectangles, and can consume and produce tokens, representing a state change. Arcs are drawn as directed edges and connect places and transitions to represent a flow relationship. A marked Petri net is thus a pair \((P,T,F,s)\) where \( P \) is a finite set of places; \( T \) is a finite set of transitions such that \( P \cap T = \emptyset \); \( F \subseteq (P \times T) \cup (T \times P) \) is a finite set of directed arcs; and \( s \in P \mapsto \mathbb{N} \) is a bag over \( P \) denoting the marking of the net [2].

Petri nets are bipartite directed graphs, meaning that each arc must connect a transition to a place, or a place to a transition. Transitions can be labeled or not. Unlabeled transitions are called silent transitions, different transitions with the same label are called duplicate transitions.

A particular class of Petri nets that has been used to model processes are workflow nets [28]. Workflow nets model individual process instances in isolation. A Petri net \((P,T,F)\) is a workflow net iff: \( P \) contains a source place \( i \) that has no incoming arcs, a sink place \( o \) that has no outgoing arcs, and the short circuited net \((P,T \cup \{t\},F \cup \{(o,t),(t,i)\})\) with \( t \) an additional transition \( \notin T \cup P \) has a flow relationship that transitively connects every place or transition to every other place or transition (strong connectedness).

3.3 Artificial Negative Event Generation in AGNesMiner

Since we extend the negative event generation algorithm as introduced in AGNesMiner [10], we repeat the original version of the algorithm below for the sake of completeness.

A high level overview of the original 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, parallelism and locality information is derived from these frequent temporal constraints. Next, the technique stipulates that the event log is made more compact, by grouping process traces that have identical sequences into grouped process instances, so that searching for similar behavior in the event log can be performed more efficiently. Finally, all negative events are induced for each grouped process instance.

### 3.3.1 Step 1: Mining Frequent Temporal Constraints

Frequent temporal constraints are constraints that hold in a sufficient number of sequences \( \sigma \) 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:

\[
\begin{align*}
\text{Existence}(1, a, \sigma) & \iff \exists x \in \sigma : \text{Activity}(x) = a \\
\text{Absence}(2, a, \sigma) & \iff \exists x, y \in \sigma : x \neq y \land \\
& \quad \text{Activity}(x) = a \\
& \quad \text{Activity}(y) = a \\
\text{Ordering}(a, b, \sigma) & \iff \exists x, y \in \sigma : x < y \land \\
& \quad \text{Activity}(x) = a \\
& \quad \text{Activity}(y) = b \\
\text{Precedence}(a, b, \sigma) & \iff \forall y \in \sigma : \text{Activity}(y) = b, \\
& \quad \exists x \in \sigma : \text{Activity}(x) = a \land x < y \\
\text{Response}(a, b, \sigma) & \iff \forall x \in \sigma : \text{Activity}(x) = a, \\
& \quad \exists y \in \sigma : \text{Activity}(y) = b \land x < y \\
\text{ChainPrec}(a, b, \sigma) & \iff \forall y \in \sigma : \text{Activity}(y) = b, \\
& \quad \exists x \in \sigma : \text{Activity}(x) = a \land x, y \\
\text{ChainResp}(a, b, \sigma) & \iff \forall x \in \sigma : \text{Activity}(x) = a, \\
& \quad \exists y \in \sigma : \text{Activity}(y) = b \land x, y \\
\text{ChainSeq}(a, b, c, \sigma) & \iff \exists x, y, z \in \sigma : x, y, z \land \\
& \quad \text{Activity}(x) = a \land \\
& \quad \text{Activity}(y) = b \land \\
& \quad \text{Activity}(z) = c
\end{align*}
\]

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:

\[
\text{Support}_{\sigma \in L}(C, L) = \frac{|S|}{|T|}
\]

for which \( S \) is a set containing the sequences \( \sigma \) for which \( C \) succeeds, and \( T \) is a set containing all sequences \( \sigma \in L \). 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:

\[
\text{Support}_{\sigma \in L}(C \rightarrow D, L) = \frac{|S \cap D|}{|T|}
\]

The confidence of a temporal association rule is defined as:

\[
\text{Conf}_{\sigma \in L}(C \rightarrow D, L) = \frac{|S \cap D|}{|S|} \times \frac{|S|}{|T|}
\]

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. For instance, it can be the case that the temporal constraint \( \text{ChainResp}(a, b, \sigma) \) has a large support, while in reality \( \langle a, b \rangle \) does not frequently occur in \( L \). Consider for an example an event log containing a large number of traces that do not contain activity type \( a \). When checking the rule \( \text{ChainResp}(a, b, \sigma) \) 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 \( \text{ChainResp}(a, b) \) 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 \( \text{Existence}(1, a, \sigma) \rightarrow \text{ChainResp}(a, b, \sigma) \). The following frequent temporal association rules are derived from an event log \( L \) (left implicit):

![Petri net representation of an ordering process.](image-url)
Absence(2, a) ⇐∀a ∈ A:

\[ \text{Supp}_{\sigma \in L}(\text{Absence}(2, a, \sigma), L) \geq t_{\text{absence}} \]

Ordering(a, b) ⇐∀a, b ∈ A:

\[ \text{Supp}_{\sigma \in L}(\text{Ordering}(a, b, \sigma), L) \geq t_{\text{ordering}} \]

Precedence(a, b) ⇐∀a, b ∈ A:

\[ \text{Conf}_{\sigma \in L}(\text{Existence}(1, b, \sigma) \rightarrow \text{Precedence}(a, b, \sigma), L) \geq t_{\text{succession}} \]

Response(a, b) ⇐∀a, b ∈ A:

\[ \text{Conf}_{\sigma \in L}(\text{Existence}(1, a, \sigma) \rightarrow \text{Response}(a, b, \sigma), L) \geq t_{\text{succession}} \]

ChainPrec(a, b) ⇐∀a, b ∈ A:

\[ \text{Conf}_{\sigma \in L}(\text{Existence}(1, b, \sigma) \rightarrow \text{ChainPrec}(a, b, \sigma), L) \geq t_{\text{succession}} \]

ChainResp(a, b) ⇐∀a, b ∈ A:

\[ \text{Conf}_{\sigma \in L}(\text{Existence}(1, a, \sigma) \rightarrow \text{ChainResp}(a, b, \sigma), L) \geq t_{\text{succession}} \]

ChainSeq(a, b, c) ⇐∀a, b, c ∈ A:

\[ \text{Supp}_{\sigma \in L}(\text{ChainSeq}(a, b, c, \sigma), L) \geq t_{\text{triplet}} \]

In the above definitions, the predicates Ordering(a, b) and ChainSeq(a, b, c) do not account for the mutual distribution of the activity types a, b and c. They remain therefore sensitive to their frequency of occurrence.

### 3.3.2 Step 2: Deriving Structural Information: Parallelism and Locality

Now that a set of temporal frequent constraints is constructed, it becomes possible to derive information about parallelism and locality of pairs of activities. Intuitively, parallelism between two activities a, b ∈ A can be assumed when it is the case in the event log L that a frequently follows b and b frequently follows a. However, in the case of multiple activities, a triple chain sequence ⟨a, b, a⟩ or ⟨b, a, b⟩ could occur in serial, without the presence of parallelism. Therefore, we require these triple chain sequences not to occur, unless it can be detected that a and b are actually part of two separate, parallel length-one loops (see Figure 3). This is the case when sequences like ⟨a, a, b⟩, ⟨b, a, a⟩, ⟨b, b, a⟩ or ⟨a, b, b⟩ are also frequent. Parallel(a, b) can consequently be defined as follows:

\[ \forall a, b \in A : (\text{ChainPrec}(a, b) \lor \text{ChainResp}(a, b)) \land \\
(\text{ChainPrec}(b, a) \lor \text{ChainResp}(b, a)) \land \\
(\sim \text{ChainSeq}(a, b, a) \land \sim \text{ChainSeq}(b, a, b) \lor \\
\text{ParallelLoop}(a, b)) \Rightarrow \text{Parallel}(a, b) \]

\[ \forall a, b \in A : \text{Parallel}(a, b) \Rightarrow \text{Parallel}(b, a) \]

\[ \forall a, b \in A : \text{ChainSeq}(a, a, a) \lor \text{ChainSeq}(b, a, a) \\
\lor \text{ChainSeq}(a, b, b) \lor \text{ChainSeq}(b, b, a) \\
\Rightarrow \text{ParallelLoop}(a, b) \]

Locality (serial occurrence denoting an explicit dependency) between activities can also be derived from the frequent temporal constraints. As a rule of thumb, a is local to b if a is frequently followed by b or b is frequently preceded by a:

\[ \forall a, b \in A : \text{ChainPrec}(a, b) \land \sim \text{Parallel}(a, b) \Rightarrow \text{Local}(a, b) \]

\[ \forall a, b \in A : \text{ChainResp}(a, b) \land \sim \text{Parallel}(a, b) \Rightarrow \text{Local}(a, b) \]

Unlike parallelism, locality is not symmetric, i.e. Local(a, b) does not imply Local(b, a). In the presence of parallelism, it is highly likely that not all local relationships can be detected. Consider the situation in Figure 4(a). Although a is local to b it is possible that the sequence ⟨a, b⟩ occurs infrequently in the event log, due to the parallelism between b and c causing sequence ⟨a, c, b⟩ to occur far more than ⟨a, b, c⟩. In such cases, however, Local(a, b) can be derived from the parallelism of b and c. Figure 4(b) on the other hand depicts a situation in which such an inference cannot be made. These concerns are translated in the following derivation rule:
\[ \forall a,b \in A : \exists c \in A : \text{Parallel}(b,c) \land \sim \text{Parallel}(a,c) \land \\
\sim \text{Parallel}(a,b) \land (\text{ChainPrec}(a,c) \lor \text{ChainResp}(a,c)) \land \\
\sim \text{Parallel}(d,b) \land (\text{ChainResp}(d,b) \lor \text{ChainPrec}(d,b)) \land \\
\exists d \in A : (\text{ChainResp}(a,d) \lor \text{ChainPrec}(a,d)) \land \\
\sim \text{Parallel}(a,d) \land \\
\Rightarrow \text{Local}(a,b) \]

The same reasoning for an AND-split can be applied to an AND-join, as displayed in Figure 4(c, d), leading to the following second derivation rule:

\[ \forall a,b \in A : \exists c \in A : \text{Parallel}(a,c) \land \sim \text{Parallel}(b,c) \land \\
\sim \text{Parallel}(a,b) \land (\text{ChainPrec}(c,b) \lor \text{ChainResp}(c,b)) \land \\
\sim \text{Parallel}(a,d) \land \\
\Rightarrow \text{Local}(a,b) \]

### 3.3.3 Step 3: Generating Artificial Negative Events

Once parallelism and locality information is derived, negative examples can be introduced in grouped process instances by checking at any given positive event whether any other activity type in the event log could occur as an event at this position. To address the problematic nature of the completeness assumption of an event log under recurrent behavior (loops), a window size parameter `windowSize` is introduced to limit the number of events which are compared when evaluating a candidate negative event. 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 probable 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.

Concurrent behavior (parallelism) can pose problems with regards to the completeness assumption of an event log as well. This issue is addressed by exploiting the previously mined parallelism information to generate parallel variants of a subsequence at hand which is used to disprove a particular candidate negative event. Taking the parallelism information into account results in a higher number of subsequences which will be used when evaluating a negative event and thus results in a weaker completeness assumption. The calculation of parallel variants is illustrated in Figure 5.

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 \( \tau_i \), it is examined which negative events can be introduced at this position. Algorithm 2 describes the pseudo-code for this process. In a first step, the event log is made more compact by

![Fig. 4. Deriving locality from parallelism.](image-url)

![Fig. 5. Calculating parallel variants.](image-url)
In this section, we present a number of improvements and modifications in order to create an artificial negative event generator to enhance process logs with negative event examples. We aim to make the completeness assumption more configurable and the generation procedure more robust so that the introduction of falsely induced 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).

We describe four extensions in this section. First, we start by introducing some new frequent temporal and structural constraints which will be used by subsequent extensions, and we modify the Ordering and ChainSeq temporal association rules so that the detection of these patterns is no longer sensitive to the frequency of occurrence of a particular activity type in the event log.

Second, we show that the original window size parameter does not suffice to counter complexity caused by recurrent behavior (loops) and thus extend the variant calculation technique to also include variants based on grouping process instances that have identical sequences $\sigma \in L$ into grouped process instances $\tau \in M$. Note that we assume all events in the event log $L$ to represent a complete life cycle transition, for example by executing a pre-filtering task on a given event log. In the next step, all negative events are induced for each grouped process instance $\tau_i$ (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. For each positive event $x_k \in \tau_i$, it is tested whether there exists a similar sequence $\tau_{j} \in AllParallelVariants(\tau_i) : \tau_{j} \not= \tau_{i}$ in the event log in which at that point an event $y_k$ has taken place that is similar to $z_k$, with $AllParallelVariants(\tau)$ a function which returns the set $\tau^P$, containing all parallel variants of sequence $\tau$. 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 $\tau_i$.

Finally, the negative events in the grouped process instances are used to induce negative events into the similar non-grouped sequences. If a grouped sequence $\tau$ contains negative events at position $k$, then the similar ungrouped sequence $\sigma$ contains corresponding negative events at position $k$. Usually, a large number of negative events can be generated, so that a probability $\pi$ is introduced as a threshold for injecting negative events into the ungrouped sequences.

Figure 6 illustrates how in an event log of two (grouped) sequences $\tau_1$ and $\tau_2$ artificial events can be generated. Note that history-dependent processes generally will require a larger window size to correctly detect all non-local dependencies. In Figure 6 for example, an unlimited window size is used. Should the window size be limited to 1, it would no longer be possible to take into account the non-local dependency between activity pairs $b - f$ and $c - g$. This underlines the tradeoff as discussed before in the introduction. Using a higher window size leads to the generation of more valuable negative events (either for discovery and evaluation purposes), 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. The improvements presented herafter aim to tackle this tradeoff, by making the completeness assumption more configurable and by introducing a number of techniques which allow to generate a set of negative events which is more likely to be both correct and complete.

### 4 IMPROVEMENTS

<table>
<thead>
<tr>
<th>Parallel variants:</th>
<th>Negative events:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>${a, b, d, e, f, h}$</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>${a, c, e, d, g, h}$</td>
</tr>
<tr>
<td>$\tau_1^P$</td>
<td>${a, b, e, d, f, h}$</td>
</tr>
<tr>
<td>$\tau_2^P$</td>
<td>${a, c, e, d, g, h}$</td>
</tr>
</tbody>
</table>

Given:

| $\tau_1$ | $\{a, b, d, e, f, h\}$ |
| $\tau_2$ | $\{a, c, e, d, g, h\}$ |

Negative events:

| $\tau_1$ | $\{a, b, d, e, f, h\}$ |
| $\tau_2$ | $\{a, c, e, d, g, h\}$ |

*: Negative events based on non-local dependencies. **: Incorrectly generated negative events.
Algorithm 2 Generating artificial negative events – original AGNEsMiner [10] implementation.

- group similar sequences \( \sigma \in L \) into \( \tau \in M \)

\( M := \text{GroupLog}(L) \)

- generate artificial events

\( N := \emptyset \)

for each \( \tau_i \in M \) do

for each \( x_k \in \tau_i \) do

\( k := \text{Position}(x_k, \tau_i) \)

for each \( b \in A \setminus \{ \text{Activity}(x_k) \} \) do

if \( \parallel \tau_j \parallel \in \text{AllParallelVariants}(\tau_j) : \forall \tau_j \in M \land \tau_j \neq \tau_i \land \forall y_j \in \tau_j \parallel \), Position\( (y_j, \tau_j \parallel) = l = \text{Position}(x_i, \tau_i) \),

\( k \text{- windowSize} < l < k : \text{Activity}(y_j) = \text{Activity}(x_i) \land y_j \in \tau_j \parallel \), Position\( (y_k, \tau_k \parallel) = k \), Activity\( (y_k) = b \)

then \( z_k := \text{Event}(x, \text{Case}(\tau_i), b, \text{completeRejected}, k) \)

- with \( x \) a new event identifier

\( N := N \cup \{ \text{NegativeEvent}(z_k, k, \tau_i) \} \)

- induce negative events in non grouped sequences \( \sigma \in L \) with injection frequency \( \pi \)

\( L_n := \text{InduceEvents}(N, L, \pi) \)

Loops which are discovered in the event log.

Since processes can combine concurrent and recurrent behavior in a complex manner, it is possible that not all variants can be calculated, or that generating all variants becomes computationally infeasible, due to the recursive and exponential nature of the problem. Therefore, the third improvement introduces a dynamic window, to make the disprove procedure for a candidate negative event more strict. Using this dynamic window with a window size of 1 weakens the completeness assumption to that of the \( \sigma \)-algorithm – meaning that an event log which contains every possible binary sequences of activities somewhere in its traces suffices to generate a fully correct set of negative events, even when no parallel or looped variants are calculated.

Finally, we introduce a technique to generate negative events without performing a similarity check between traces in the event log, by relying only on discovered activity type dependency information.

4.1 Improvement 1: New Temporal Constraint Confidence Measures

We redefine the \text{Ordering} and \text{ChainSeq} temporal association rules so that the detection of these patterns is no longer sensitive to the frequency of occurrence of particular activity types. The temporal constraints describing \text{Ordering} and \text{ChainSeq} are redefined as follows:

\( \text{Ordering}(a, b, c, \sigma) \Leftrightarrow \)

\( \forall x, y \in \sigma, \text{Activity}(x) = a, \text{Activity}(y) = b : x \neq y \)

\( \text{ChainSeq}(a, b, c, \sigma) \Leftrightarrow \)

\( \forall z \in \sigma : \text{Activity}(z) = c, \exists x, y \in \sigma : \text{Activity}(x) = a \land \text{Activity}(y) = b \land x \neq y \lor \exists x, z \in \sigma : \text{Activity}(x) = a \land \text{Activity}(z) = c \land x \neq z \lor \exists y, z \in \sigma : \text{Activity}(y) = b \land \text{Activity}(z) = c \land x \neq y \land y \neq z \)

The corresponding association rules are now:

\( \text{Ordering}(a, b) \Leftrightarrow \forall a, b \in A : \)

\( \text{Conf}_{\sigma \in L}(\text{Existence}(1, a, \sigma) \land \text{Existence}(1, b, \sigma)) \rightarrow \text{Ordering}(a, b, \sigma, L) \geq l_{\text{succession}} \)

\( \text{ChainSeq}(a, b, c) \Leftrightarrow \forall a, b, c \in A : \)

\( \text{Conf}_{\sigma \in L}(\text{Existence}(1, a, \sigma) \lor \text{Existence}(1, b, \sigma) \lor \text{Existence}(1, c, \sigma)) \rightarrow \text{ChainSeq}(a, b, c, \sigma, L) \geq l_{\text{triple}} \)

We also define the following additional temporal constraints:

\( \text{Iteration}(l, u, v, \sigma) \Leftrightarrow \forall a, b \in A : \)

\( \exists s \in \sigma : \text{Position}(s, \sigma) = l \land \forall i \in [1, l], j \in [1, u] : \)

\( \forall x \in s, y \in v, i = \text{Position}(y, v), t + i + j l = \text{Position}(x, \sigma) : \)

\( \text{Activity}(x) = \text{Activity}(y) \land l = |v|, v \subseteq \sigma, 1 = \text{Position}(u, v) \)
StartActivity\((a, \sigma)\) ⇔ ∀x ∈ σ, Activity\(x) = a : \neg Position(x, \sigma) = 1$

EndActivity\((a, \sigma)\) ⇔ ∀x ∈ σ, Activity\(x) = a : \neg Position(x, \sigma) = |\sigma|

With the corresponding association rules:

\[\text{Iteration}(l, n, v) ⇔ ∀v ⊆ σ, σ ∈ L, l = |v|, n = 2, 0 < l ≤ 3 : \neg Conf_{σ ∈ L}(∀x ∈ v)

Existence\(1, Activity(x), σ\) → Iteration\((l, n, v, σ)\) \geq t_{iteration}\]

\[\text{StartActivity}(a) ⇔ ∀a ∈ A : Conf_{σ ∈ L}(\exists Activity(x), σ) \rightarrow \neg StartActivity(a, σ) \geq t_{position}\]

\[\text{EndActivity}(b) ⇔ ∀a ∈ A : Conf_{σ ∈ L}(\exists Activity(x), σ) \rightarrow \neg EndActivity(a, σ) \geq t_{position}\]

The Iteration temporal constraint holds in a sequence σ if an activity sequence v with length l iterates n times in σ. In order to keep to 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 ≤ 3.

4.2 Improvement 2: Variant Calculation with Loop Discovery

The Iteration temporal constraint as defined in the previous section allows us to formulate a loop discovery heuristic. Loops, however, 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\(3, 2, (b, c, d)\), Iteration\(3, 2, (c, d, b)\) and Iteration\(3, 2, (d, b, c)\). 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:

\[∀v \subseteq σ, σ ∈ L, l = |v| : \neg Iteration(l, v) \land \exists a ∈ A : \neg Local(a, u) \rightarrow Loop(v)\]

Based on the structural Loop constraints, the variant calculation method can be extended to take recurrent behavior into account as well, next to concurrent behavior. Figure 7 depicts an example of the generation of loop variants. The variant construction method works as follows: for each activity in a sequence σ, it is investigated whether it is possible to insert additional or new loop iterations at this position (by looking at discovered Local and Loop constraints), and whether already present loop iterations can be removed. Since the addition of loop iterations could potentially lead to infinitely long sequences, some additional parameters have to be defined to guide to construction of loop variants. Firstly, in the case of inserting additional loop iterations, a parameter loopAddUntil defines the upper limit for the number of iterations in each loop. Setting this parameter to unlimited does not lead to generation of infinitely long sequences, but only adds enough iterations of each loop to each sequence so that for every candidate negative event considered, a loop variant trace can always be found where the window to be checked always contains loop iterations which could occur up until that point. Furthermore, a boolean parameter loopAddFromZero governs if loop iterations are to be added at insert positions where no iterations of the considered loop are yet present. Secondly, for the removal of iterations, a parameter loopRemoveUntil defines the lower limit for the number of iterations in each loop. Setting this parameter to zero completely removes loops where possible. A large number of variants can often be calculated for given sequences in an event log, leading to a higher number of traces which will be used when evaluating a negative event, thus resulting in less incorrect cases.

Note that, instead of having to manually specify a value for loopAddUntil and loopRemoveUntil, it is also possible to set these values based on the minimum and maximum number of subsequent iteration occurrences within a sequence for each discovered loop. As such, for each discovered Loop\(v\) constraint, we can also define corresponding MinLoopOccurence\(i, v\) and MaxLoopOccurence\(j, v\) constraints. The advantage of using this information for the calculation of loop variants is that each distinct loop can now be grown or shrunk according to the real behavior present in the log (instead of using global parameters over all loops), but note that this makes the completeness assumption more strict in the sense that the minimum and maximum possible iterations of each loop should indeed be present in the given event log.

Figure 8 depicts an example in order to demonstrate the usefulness of loop variant calculation when generating negative events. Although the window size parameter has been set to 1 for the generation of negative events in order to handle recurrent behavior, incorrect cases are still introduced in the event log due to the strict position comparison between the window of the positive trace at hand and the window of the trace used to disprove the negative event currently under consideration, denoted by Position\(y_k, \tau_1\) = \(y_k, \tau_2\) in Algorithm 2. It is this strict comparison which causes the interesting dissimilarity in the set of negative events for \(\tau_1\) and \(\tau_2\) in Figure 8: whereas \(\tau_1\) does not include \(b\) as a negative event in the previous-to-last position (at the position of activity \(c\)), \(\tau_2\) does falsely introduce \(b\) as a
negative example at the position of activity \(d\), due to the difference in lengths of the two sequences caused by the loop. Furthermore, the original algorithm is not able to discover the recurrent behavior in the underlying process, leading to the rejection of completion of \(c\) and \(d\) at each position where \(b\) occurs. Using the loop variant calculation explained above avoids the generation of these incorrect cases. Moreover, it is no longer required to use the lowest possible window (size 1), so that non-local, history-dependent behavior – if present – can again be captured by the generated negative events.

Finally, note that the generation of incorrect examples is not completely avoided when using loop variant calculation in the example of Figure 8. \(\tau_j\), for example, still rejects the completion of \(c\) activities after the completion of \(a\). This is not due to a particular intricacies of loop calculation (indeed, similar behavior can be seen in Figure 6) for the parallel variants, but rather to the following condition in Algorithm 2: \(\tau_j \neq \tau_i\). This condition prohibits variants of the current trace under consideration to be used as disproving traces for a particular candidate negative event. As such, we are able to introduce a boolean parameter \(\text{useOwnVariants}\) so that the above fragment can be reformulated as \(\tau_j^{||} \neq \tau_i\), leading to the removal of all incorrect negative events. Since most real life event logs contain enough distinct grouped traces, however, this issue should not commonly occur in practice.

negative example at the position of activity \(d\), due to the difference in lengths of the two sequences caused by the loop. Furthermore, the original algorithm is not able to discover the recurrent behavior in the underlying process, leading to the rejection of completion of \(c\) and \(d\) at each position where \(b\) occurs. Using the loop variant calculation explained above avoids the generation of these incorrect cases. Moreover, it is no longer required to use the lowest possible window (size 1), so that non-local, history-dependent behavior – if present – can again be captured by the generated negative events.

Finally, note that the generation of incorrect examples is not completely avoided when using loop variant calculation in the example of Figure 8. \(\tau_j\), for example, still rejects the completion of \(c\) activities after the completion of \(a\). This is not due to a particular intricacies of loop calculation (indeed, similar behavior can be seen in Figure 6) for the parallel variants, but rather to the following condition in Algorithm 2: \(\tau_j \neq \tau_i\). This condition prohibits variants of the current trace under consideration to be used as disproving traces for a particular candidate negative event. As such, we are able to introduce a boolean parameter \(\text{useOwnVariants}\) so that the above fragment can be reformulated as \(\tau_j^{||} \neq \tau_i\), leading to the removal of all incorrect negative events. Since most real life event logs contain enough distinct grouped traces, however, this issue should not commonly occur in practice.
\(\langle a, b, c, d \rangle\) or \(\langle a, c, b, d \rangle\) subsequences to repeat two times in a sequential manner. The same problem exists when loops are nested in other loops. Thirdly, even when all constructs could be detected – or given – in a correct manner, the generation of variants still remains an exponentially complex problem, meaning that a rise in a number of concurrency or recurrence constructs more than proportionally increases the number of variants which can be generated, especially when considering the possibility that parallel variants could be calculated for previously calculated loop variants, loop variants for previously calculated parallel variants, and so on, making the variant generation method a recursive problem as well.

Instead of trying to deal with the above problems (e.g. by adjusting the Loop and Iteration constraints to take into account concurrency), we limit the variant calculation to the simple heuristics as described above, with a maximum loop detection length of 3, and a variant calculation scheme where the parallel variants are calculated first and loop variants are generated afterward over this set; \(\text{AllVariants}(\tau) = \text{AllLoopVariants}(\text{AllParallelVariants}(\tau))\). To deal with the problem of complex constructs leading to possible incorrect negative events, we adjust the window based algorithm to remove the strict position requirement: \(\text{Position}(y_k \tau_k^\sigma) = k\) in Algorithm 2. Instead, we compare the window of the original “positive” trace with each possible window in the “candidate disproving” trace \(\tau_{\sigma}^\tau\), 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 \(\sigma = \langle a, b, c, d, f \rangle\) and \(\tau = \langle a, b, c, e, c, f \rangle\) a candidate disproving trace under consideration. Assume we are currently checking to see if candidate activity \(e\) could also occur instead of \(d\) in \(\sigma\). In cases where a strict window is used, with size equal to, say, 2, the candidate disproving trace \(\tau\) fails to disprove the negative event, since \(\text{Position}(d, \sigma) \neq \text{Position}(e, \tau)\). Instead of doing so, we now compare the window \(\langle b, c \rangle\) with each possible window in \(\tau\). In this case, the window of size 2 before activity \(e\) in \(\tau\) is also equal to \(\langle b, c \rangle\). This leads to a correct rejection of the candidate negative event. The modified algorithm using a dynamic window is listed in Algorithm 3.

Note that, depending on the window size configured, cases might now exist where the size of the window in the original trace \(\sigma\) is unequal to the size of a window in a candidate disproving trace \(\tau\). In cases where the window in \(\tau\) is larger than the window in \(\sigma\), an effective window with size equal to the window used in \(\sigma\) is used. When the window in \(\tau\) is smaller than the window in \(\sigma\), 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 \(\sigma = \langle a, b, c, d, f \rangle\) and \(\tau = \langle a, b, c, e, f \rangle\) 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 \(\sigma\). Based on this information, two windows can be defined in \(\tau\) 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 \(\sigma\). On the other hand, the other window \(\langle a \rangle\) is smaller than the window in \(\sigma\), 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 \(\text{minimumWindowSize}\) to denote a minimum required length for a window to be used in the negative event rejection procedure. Setting this parameter to -1 denotes that the window in \(\tau\) should be at least of the same size as the current window in \(\sigma\).

Using a dynamic window now allows us to dramatically 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.

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.

### 4.4 Improvement 4: Dependency Based Negative Event Generation

Instead of using a window based trace comparison algorithm, an alternative way of generating artificial negative events is to directly use the discovered structural information from the temporal frequent constraints and association rules. Locality, (long-distance) dependency and implicit dependency information can thus be applied towards the induction of negative events.

#### 4.4.1 Locality (or: Explicit Dependencies)

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 when this activity type is not locally dependent on the activity type of the event completed.
Algorithm 3 Artificial event generation algorithm with dynamic window.

```plaintext
- group similar sequences \( \sigma \in L \) into \( \tau \in M \)
- generate artificial events
\( N := \emptyset \)
for each \( \tau_i \in M \) do
  for each \( x_k \in \tau_i \) do
    \( k := \text{Position}(x_k, \tau_i) \)
    for each \( b \in A \setminus \{ \text{Activity}(x_k) \} \) do
      if \( \exists \tau_j \in \text{AllVar} \text{iants}(\tau_j) : \forall \tau_j \in M \land \tau_j \neq \tau_i \land \forall y_k \in \tau_j \setminus \tau_i, j = \text{Position}(y_k, \tau_j), \text{Activity}(y_k) = b : \forall y_l \in \tau_j, j - 1 = \text{Position}(y_l, \tau_j), k - l = \text{Position}(x_k, \tau_i), 0 < l \leq \text{windowSize} : \text{Activity}(y_l) = \text{Activity}(x_1) \)
      then \( z_k := \text{Event}(x, \text{Case}(\tau_i), b, \text{completeRejected}, k) \)
      -- with \( x \) a new event identifier
      \( N := N \cup \{ \text{NegativeEvent}(z_k, k, \tau_i) \} \)
- induce negative events in non grouped sequences \( \sigma \in L \) with injection frequency \( \pi \)
\( L_n := \text{InduceEvents}(N, L, \pi) \)
```

at the previous position:

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

Instead of using \( \text{ChainPrec}(a, c) \lor \text{ChainResp}(a, c) \) in the derivation rule for \( \text{Local} \), we can also define a \( \text{StrongLocal} \) variant using \( \text{ChainPrec}(a, c) \land \text{ChainResp}(a, c) \) instead. Using \( \text{StrongLocal} \) instead of \( \text{Local} \) in the rule above leads to a weaker check before generating a negative event, since, generally, less \( \text{StrongLocal} \) constructs can be discovered than \( \text{Local} \). Remark also that the construction of \( \text{Local} \) predicates takes parallelism information into account.

4.4.2 Long-Distance (with: Implicit) Dependencies

We define the following structural derivation rules to mine all dependencies between activity types (both implicit and explicit), similar to \( \text{Local} \) (explicit dependencies only):

\[
\forall a, b \in A : (\text{Precedence}(a, b) \lor \text{Response}(a, b)) \land \sim \text{Parallel}(a, b) \Rightarrow \text{Dependence}(a, b)
\]

\[
\forall a, b \in A : (\text{Precedence}(a, b) \land \text{Response}(a, b)) \land \sim \text{Parallel}(a, b) \Rightarrow \text{StrongDependence}(a, b)
\]

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.

Based on these dependencies, the rule of thumb defined above can be expanded: negative events with a certain activity type that are locally dependent on the activity type of the event completed at the previous position can also be added before a given completed event, when not all activities on which the negative event under consideration is strongly dependent on were completed before.

\[
\forall a, b \in A : \text{StrongDependence}(a, b) \land \sim \text{Local}(a, b) \Rightarrow \text{LongDistanceDependence}(a, b)
\]

\[
\forall \sigma \in M, 1 < k \leq |\sigma|, k = \text{Position}(x_k, \sigma), \ a \in A, a \neq \text{Activity}(x_k), k - 1 = \text{Position}(x_{k-1}, \sigma), \ v = \{ i \in \sigma | \text{Position}(i, \sigma) < k \} : \text{Local}(\text{Activity}(x_{k-1}), a) \land \exists b \in A, b \notin v : \text{ImplicitDependence}(b, a) \Rightarrow \text{Event}(x, \text{Case}(x_k), a, \text{completeRejected}, k)
\]

Note finally that we can also use the \( \text{LongDistanceDependence} \) construct to suggest an optimal minimum window size, i.e. a window size which is able to capture all pairs of activity types between which an implicit long-distance dependency relation exists. To do so, we need to restrict \( \text{LongDistanceDependence} \) a bit further to drop the long-distance dependencies which do not correspond with an implicit dependency in the underlying process model. For example, a \( \text{LongDistanceDependence} \) construct can always be found between starting and ending activities. However, a starting activity is always followed by the ending activity, so that this long-distance 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 long-distance dependencies (derived with the
for the various parameter configurations with two given parameter setting configurations. We compare the results of the artificial event generation 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 to 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 “Minimal 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 $\text{MinimalWindowSize}(M)$) 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 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 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.

5 EXPERIMENTAL RESULTS AND DISCUSSION

We have implemented our revised artificial negative event generation technique in ProM 6 [16]. We test the improvements above with the “DriversLicenseLoop” log, an artificial process log which has been used before by de Medeiros et al. [5] 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 to 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 “Minimal 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 $\text{MinimalWindowSize}(M)$) 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 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 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

A large number of algorithms have been developed to discover sequential control-flow models from event logs.
Table 1
Used parameter setting configurations for the artificial event generation tests.

<table>
<thead>
<tr>
<th>Parameter Configuration Identifier</th>
<th>Window Size</th>
<th>Dynamic Window Size</th>
<th>Minimum Window Size</th>
<th>Structural Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>originalWs-1</td>
<td>yes</td>
<td>no</td>
<td>–</td>
<td>no</td>
</tr>
<tr>
<td>originalWs3</td>
<td>yes</td>
<td>3</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>originalWs1</td>
<td>yes</td>
<td>1</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>dynamicWs-1MinWs-1</td>
<td>yes</td>
<td>-1</td>
<td>yes</td>
<td>-1</td>
</tr>
<tr>
<td>dynamicWs3MinWs-1</td>
<td>yes</td>
<td>3</td>
<td>yes</td>
<td>1 no</td>
</tr>
<tr>
<td>dynamicWs1MinWs-1</td>
<td>yes</td>
<td>1</td>
<td>yes</td>
<td>1 no</td>
</tr>
<tr>
<td>dynamicWs-1MinWs1</td>
<td>yes</td>
<td>3</td>
<td>yes</td>
<td>1 no</td>
</tr>
<tr>
<td>dynamicWs3MinWs1</td>
<td>yes</td>
<td>1</td>
<td>yes</td>
<td>1 no</td>
</tr>
<tr>
<td>dynamicWs1MinWs1</td>
<td>yes</td>
<td>1</td>
<td>yes</td>
<td>1 no</td>
</tr>
<tr>
<td>strucOnly</td>
<td>no</td>
<td>–</td>
<td>–</td>
<td>yes</td>
</tr>
<tr>
<td>strucNonDynamicWs-1</td>
<td>yes</td>
<td>-1</td>
<td>no</td>
<td>– yes</td>
</tr>
<tr>
<td>strucNonDynamicWs3</td>
<td>yes</td>
<td>3</td>
<td>no</td>
<td>– yes</td>
</tr>
<tr>
<td>strucNonDynamicWs1</td>
<td>yes</td>
<td>1</td>
<td>no</td>
<td>– yes</td>
</tr>
<tr>
<td>strucDynamicWs-1MinWs-1</td>
<td>yes</td>
<td>-1</td>
<td>yes</td>
<td>-1 yes</td>
</tr>
<tr>
<td>strucDynamicWs3MinWs-1</td>
<td>yes</td>
<td>3</td>
<td>yes</td>
<td>1 yes</td>
</tr>
<tr>
<td>strucDynamicWs1MinWs-1</td>
<td>yes</td>
<td>1</td>
<td>yes</td>
<td>1 yes</td>
</tr>
<tr>
<td>strucDynamicWs-1MinWs1</td>
<td>yes</td>
<td>1</td>
<td>yes</td>
<td>1 yes</td>
</tr>
<tr>
<td>strucDynamicWs3MinWs1</td>
<td>yes</td>
<td>1</td>
<td>yes</td>
<td>1 yes</td>
</tr>
<tr>
<td>strucDynamicWs1MinWs1</td>
<td>yes</td>
<td>1</td>
<td>yes</td>
<td>1 yes</td>
</tr>
</tbody>
</table>

Table 2
Results of the DriversLicenseLoop experiment under various configurations.

<table>
<thead>
<tr>
<th>Parameter Configuration Identifier</th>
<th>Incorrect Negative Events</th>
<th>Total Negative Events</th>
<th>Correctness</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully Correct Log</td>
<td>0</td>
<td>44866</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Naive Generation Method</td>
<td>2484</td>
<td>47350</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>originalWs-1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>originalWs3</td>
<td>621</td>
<td>45487</td>
<td>75,0%</td>
<td>100%</td>
</tr>
<tr>
<td>originalWs1</td>
<td>621</td>
<td>44866</td>
<td>75,0%</td>
<td>98,6%</td>
</tr>
<tr>
<td>dynamicWs-1MinWs-1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>dynamicWs3MinWs-1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>dynamicWs1MinWs-1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>dynamicWs-1MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>dynamicWs3MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>dynamicWs1MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucOnly</td>
<td>0</td>
<td>44866</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>strucNonDynamicWs-1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucNonDynamicWs3</td>
<td>621</td>
<td>45487</td>
<td>75,0%</td>
<td>100%</td>
</tr>
<tr>
<td>strucNonDynamicWs1</td>
<td>621</td>
<td>45349</td>
<td>75,0%</td>
<td>99,7%</td>
</tr>
<tr>
<td>strucDynamicWs-1MinWs-1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs3MinWs-1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs1MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs-1MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs3MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs1MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs-1MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs3MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs1MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs-1MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs3MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs1MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs-1MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs3MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
<tr>
<td>strucDynamicWs1MinWs1</td>
<td>642</td>
<td>45508</td>
<td>74,2%</td>
<td>100%</td>
</tr>
</tbody>
</table>

that allow to examine the real execution traces of business activities. However, these process discovery tasks are often confronted with some particular difficulties. One such difficulty 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 and consequently cannot guide the discovery task. In this paper, we proposed an extension for the negative event generation method as first introduced in the AGNEsMiner process discovery algorithm [10]. 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-local, history-dependent constructs. The existence of the tradeoff between these two goals is due to the completeness assumption made over an event log when generating artificial negative events. In the original version of the algorithm, a configurable completeness assumption is defined by proposing a window size parameter and a negative event injection probability. We presented several improvements to make this configured completeness assumption less strict and the artificial event generation procedure more robust in order to prevent the introduction of falsely induced
negative events in cases where an event log does not capture all possible behavior. Ensuring a correct induction of negative events proves especially helpful in practice when these events are subsequently used for evaluation purposes. The improvements entail: firstly, new definitions for temporal constraint confidence measures; second, a variant calculation technique based on recurrent behavior (short loops); third, a redefinition of the window based trace similarity check; fourth, the appliance of non-window based negative event generation techniques, based on discovered dependency information.

By means of an experiment on an artificial log, we illustrated the benefits of the proposed improvements. In future work, we aim at extending the set of artificial logs included in the experiment. Furthermore, we remarked above that the given log in our experiment gives a good coverage of all possible execution traces as allowed by the underlying process model, so that even a naive generation method performs relatively well when the ratio of incorrect negative events versus the total number of negative events is considered. Therefore, a followup study should also investigate the effects of making the given log less complete with respect to the underlying model. Finally, we also aim to apply the improved generation of negative events method towards the evaluation of real-life event logs (which is not done in the experiments, as the number of all correct negative events is then, of course, unknown). Goedertier et al. [10] have yet defined two evaluation metrics to capture recall and specificity (i.e. the amount of behavior present in the event log which is captured by the proposed process model) using negative events. De Weerdt et al. [12] additionally define a precision measure based on negative events to gauge whether a mined process model does not underfit the behavior present in the event log.

ACKNOWLEDGEMENT

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

REFERENCES


[18] D. Angluin, “Inductive inference of formal lan-


