Journal of Machine Learning Research 10 (2009) 1305-1340 Submitted 3/08; Revised 9/08; Published 6/09

Robust Process Discovery with Artificial Negative Events

Stijn Goedertier STIJN.GOEDERTIER@ECON.KULEUVEN.BE
David Martens* DAVID.MARTENS@ECON.KULEUVEN.BE
Jan Vanthienen JAN.VANTHIENEN@ECON.KULEUVEN.BE
Bart Baesens† BART.BAESENS@ECON.KULEUVEN.BE
Faculty of Business and Economics
Katholieke Universiteit Leuven
Leuven, Naamsestraat 69, Belgium

Editor: Paolo Frasconi, Kristian Kersting, Hannu Toivonen and Koji Tsuda

Abstract

Process discovery is the automated construction of structured process models from information system event logs. Such event logs often contain positive examples only. Without negative examples, it is a challenge to strike the right balance between recall and specificity, and to deal with problems such as expressiveness, noise, incomplete event logs, or the inclusion of prior knowledge. In this paper, we present a configurable technique that deals with these challenges by representing process discovery as a multi-relational classification problem on event logs supplemented with Artificially Generated Negative Events (AGNEs). This problem formulation allows using learning algorithms and evaluation techniques that are well-know in the machine learning community. Moreover, it allows users to have a declarative control over the inductive bias and language bias.

Keywords: graph pattern discovery, inductive logic programming, Petri net, process discovery, positive data only

1. Introduction

Learning descriptive or predictive models from sequence data is an important data mining task with applications in Web usage mining, fraud detection, bio-informatics, and process discovery. The learning task can be formulated as follows: given a sequence database that contains a finite number of sequences, find a useful generative model that describes or predicts its spatio-temporal properties. Depending on the application domain, a variety of sequence models and corresponding learning algorithms are available. For instance, probabilistic generative models such as (hidden) Markov models have been successfully applied in speech analysis, and bio-informatics (Durbin et al., 1998), whereas deterministic models such as partial orders have been applied in domains such as Web usage mining (Mannila and Meek, 2000; Pei et al., 2006). In this paper, we focus on the problem of process discovery, the discovery of business process models from event-based data generated by information systems. Process models typically contain structures such as sequences, or-splits, and-splits (parallel threads), or-joins, and-joins, loops (iterations), non-local, non-free, history-dependent or-splits, and duplicate activities. Because Petri nets can represent these con-

∗. David Martens is also at the Department of Business Administration and Public Management, University College Ghent, Association Ghent University, Belgium.
†. Bart Baesens is also at the School of Management, University of Southampton, United Kingdom.

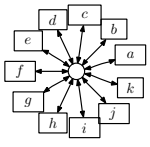
structs, they are known to be a convenient process modeling language (van der Aalst, 1998; Alves de Medeiros, 2006) provides a detailed overview of how these structures can be represented with Petri nets.

The motivation for process discovery is the abundant availability of information system event logs. The analysis of such event logs can provide insight into how processes *actually* take place, and to what extent actual processes deviate from a normative process model. Information system events keep track of, among others, the completion of an activity of a particular activity type. For example, an event can report that a particular activity of type 'apply for license' has completed. The goal of process discovery is to construct a useful process model that *describes* the event sequences recorded in the event log. Example 1, illustrates the learning problem for a fictitious driver's license application process. Given the event sequence database in Example 1, a useful process models is to be conceived.

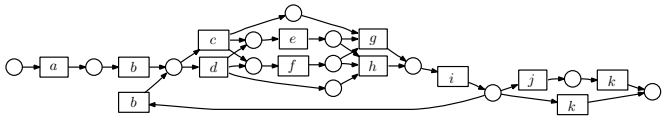
Example 1 *DriversLicensel—discovery of a driver's license application process with loop. The transitions correspond to activity types that have the following meaning: a start, b apply for license, c attend classes cars, d attend classes motor bikes, e obtain insurance, f theoretical exam, g practical exam cars, h practical exam motor bikes, i get result, j receive license, and k end.*

σ_1	*abcefgijk*
σ_2	*abdfehijk*
σ_3	*abdefhijbdfehijk*
σ_4	*abcfegik*
...	...

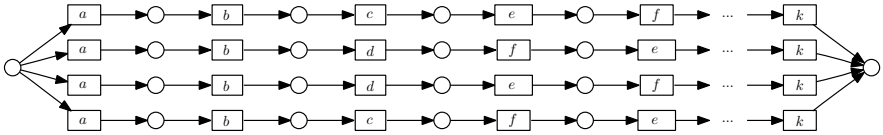
(a) sequences



(b) general



(c) specific and general



(d) over-specific

The construction of useful process models from an event log is subjected to many challenging problems. An inherent difficulty is that process discovery is limited to a setting of unsupervised learning. Event logs rarely contain negative information about state transitions that were prevented from taking place. Such negative information can be useful to discover the discriminating properties of the underlying process. In the absence of negative information, it is necessary to provide a learner with a particular *inductive bias*, to accurately strike the right balance between generality and specificity. The absence of negative information, makes the learning task aim at accurately summarizing an event log such that the discovered process model allows the observed behavior (recall) but does not include unintended, random behavior that is not present in the event log (specificity). In addition to accuracy, the learning problem faces challenges such as expressiveness, noise, incomplete event logs, and the inclusion of prior knowledge:

- **accuracy**: accuracy refers to the extent to which the induced model fits the behavior in the event log. Accuracy necessitates a tradeoff between specificity and recall. The Petri net in

Example 1, called a flower model, is capable of parsing every sequence in the event log. However, it can be considered to be overly general as it allows any activity to occur in any order. In contrast, the Petri net in Example 1 is overly specific, as it provides a mere enumeration of the different sequences in the event log. The Petri net in Example 1 is likely to be the more suitable process model. It is well-structured, and strikes a reasonable balance between specificity and generality, allowing for instance an unseen sequence *abcefgik*, but disallowing random behavior. An additional difficulty is that in the absence of negative information, the specificity of a learned process model is difficult to quantify.

- **expressiveness**: expressiveness relates to the ability to comprehensively summarize an event log using a rich palette of structures such as sequences, or-splits, and-splits (parallel threads), or-joins, and-joins, loops (iterations), history-dependent or-splits, and duplicate activities.
- **noise**: human-centric processes are prone to exceptions and logging errors. This causes additional low-frequent behavior to be present in the event log that is unwanted in the process model to be learned. Process discovery algorithms face the challenge of not overfitting this noise.
- **incomplete logs**: incomplete event logs do not contain the complete set of sequences that occur according to the underlying, real-life process. This is particularly the case for processes that portray a large amount of concurrent and recurrent behavior. In this case, process discovery algorithms must be capable of generalizing beyond the observed behavior.
- **prior knowledge**: the problem of consolidating the knowledge extracted from the data with the knowledge representing the experience of domain experts, is called the knowledge fusion problem (Dybowski et al., 2003). Prior knowledge constrains the hypothesis space of a sequence mining algorithm. In the context of process discovery, prior knowledge might refer to knowledge about concurrency (parallelism), locality, or exclusivity of activities. When a learner produces a process model that is not in line with the prior knowledge of a domain expert, the expert might refuse using the discovered process model. For instance, a domain expert might refuse a process model in which a pair of activities cannot take place concurrently, whereas in reality such parallelism is actually allowed.

In this paper, these challenges addressed by representing process discovery as an ILP classification learning problem on event logs supplemented with artificially generated negative events (AGNEs). The AGNEs technique is capable of constructing Petri net models from event logs and has been implemented as a mining plugin in the ProM framework. A benchmark experiment with 34 artificial event logs and comparison to four state-of-the-art process discovery algorithms indicate that the technique is expressive, robust to noise, and capable of dealing with incomplete event logs. A second contribution of the paper is the definition of a new metric for quantifying the specificity of an induced process model, based on these artificially generated negative events.

The remainder of this paper is structured as follows. Section 2 introduces some preliminaries and notations about inductive logic programming, event logs, and Petri nets. Section 3 explains the rationale for generating artificial negative events and provides a detailed description of the used algorithms. Section 4 discusses the process discovery technique. Section 5 introduces the new specificity metric that is based on negative events. Sections 6 and 7 provide both an experimental and practical evaluation of the process discovery technique. Finally, Section 8 provides an overview of the work in the area of process discovery and outlines the contributions made.

2. Preliminaries

This section introduces the most important concepts and notations that are used in the remainder of this paper.

2.1 Inductive Logic Programming

Inductive Logic Programming (ILP) (Muggleton, 1990; Džeroski and Lavrač, 1994, 2001; Džeroski, 2003) is a research domain in machine learning involving learners that use logic programming to represent data, background knowledge, the hypothesis space, and the learned hypothesis. ILP learners are called multi-relational learners and extend classical, uni-relational learners in the sense that they can not only learn patterns that occur within single tuples (within rows), but can also find patterns that may range over different tuples of different relations (between multiple rows of a single or multiple tables). For process discovery, this multi-relational property is much desired, as it allows discovering patterns that relate the occurrence of an event to the occurrence of any other event in the event log.

Within ILP, *concept learning* is an important learning task. An ILP classification learner will search for a hypothesis H in a hypothesis space $\mathcal{S}$ that best discriminates between the positive P and negative examples N ($E = P \cup N$) in combination with some given background knowledge B. A particularly salient feature of such learners is that they have a highly configurable language bias. The language bias $\mathcal{L}$ specifies the hypothesis space $\mathcal{S}$ of logic programs H that can be considered. In addition, users of ILP learners can specify background knowledge B as a logic program. Such background knowledge is a more parsimonious encoding of knowledge that is true about every example, than is the case for the attribute-value encoding of propositional learners. In addition to their multi-relational capabilities, the power of ILP concept learners lies with the configurability of their language bias $\mathcal{L}$ and background knowledge B. The effectiveness by which an ILP learner can be applied to a learning task depends on the choices that are made in representing the examples E, the background knowledge B and the language bias $\mathcal{L}$.

In this text, we make use of TILDE (Blockeel and De Raedt, 1998), a first-order decision tree learner available in the ACE-ilProlog data mining system (Blockeel et al., 2002). Blockeel (1998) formalizes the learning task of TILDE as follows:
given:

- a set of classes C
- a set of classified examples E, each example $(e,c) \in E$ is an independent logic program for which the predicate $Class(c)$ denotes that e is classified into class c.
- a logic program B that represents the background knowledge
- a language bias $\mathcal{L}$ that specifies a hypothesis space $\mathcal{S}$ of logic programs.

find: a hypothesis $H \in \mathcal{S}$ (a logic program) such that for all labeled examples $(e,c) \in E$,

- $\forall e \in E : H \wedge e \wedge B \models Class(c)$
- $\forall e \in E, \forall c' \in C \backslash \{c\} : H \wedge e \wedge B \not\models Class(c')$.

TILDE is a first-order generalization of the well-known C4.5 algorithm for decision tree induction (Quinlan, 1993). Like C4.5, TILDE (Blockeel and De Raedt, 1998; Blockeel et al., 2002) obtains classification rules by recursively partitioning the data set according to logical conditions that can be represented as nodes in a tree. This top-down induction of logical decision trees (TILDE) is driven by refining the node criteria according to the provided language bias $\mathcal{L}$. Unlike C4.5, TILDE is capable of inducing first-order logical decision trees (FOLDT). A FOLDT is a tree that holds logical formula containing variables instead of propositions. Blockeel and De Raedt (1998) show how each FOLDT can be converted into a logic program.

2.2 Event Logs

In process discovery, an event log is a database of event sequences. Each event reports an instantaneous state change of an activity of a particular activity type. Activities and events that pertain to the same process instance are identified by a so-called case identifier. In process discovery, the MXML format for event logs (van Dongen and van der Aalst, 2005a) is the commonly accepted format for event logs. To use event logs with TILDE event logs have to be represented as a logical program. Let X be a set of event identifiers, P a set of case identifiers, the alphabet A a set of activity types, and $E = \{completed, completeRejected\}$ a set of event types. An event predicate is a quintuple $Event(x, p, a, e, t)$, where $x \in X$ is the event identifier, $p \in P$ is the case identifier, $a \in A$ the activity type, $e \in E$ the event type, and $t \in \mathbb{N}$ the time of occurrence of the event. The function $Case \in X \cup L \rightarrow P$ denotes the case identifier of an event or a sequence. The function $AT \in X \rightarrow A$ denotes the activity type of an event. The function $ET \in X \rightarrow E$ denotes the event type of an event. The function $Time \in X \rightarrow \mathbb{N}$ denotes the time of occurrence of an event. The set X of identifiers has a complete ordering, such that $\forall x, y \in X : x < y \vee y < x$ and $\forall x, y \in L : x < y \Rightarrow Time(x) \leq Time(y)$. The event types $E = \{completed, completeRejected\}$ respectively indicate the completion of a particular activity or that the completion of a particular activity could not take place, a negative event.

Let an event log L be a set of sequences σ. Let $\sigma \in L$ be an event sequence, an ordered set of event identifiers $x \in X$ of events pertaining to the same process instance as denoted by the case id; $\sigma = \{x \mid x \in X \wedge Case(x) = Case(\sigma)\}$. The function $Position \in X \times L \rightarrow \mathbb{N}_0$ denotes the position of an event with identifier $x \in X$ in the sequence $\sigma \in L$. Two subsequent event identifiers within a sequence σ can be represented as a sequence $x.y \subseteq \sigma$. We define the .-predicate as follows

$$x.y \Leftrightarrow \exists x, y \in \sigma : x < y \wedge \nexists z \in \sigma : x < z < y.$$

In the text, this predicate is used within the context of a single sequence σ which is therefore left implicit. Given that $AT(x) = a, AT(y) = b$ the information in the sequence can be further abbreviated as *ab*, because the order of the activity types in a sequence is the most important information for the purpose of process discovery. This notation is used to represent the event log in Example 1. Each row σ_i in the event log represents a different execution sequence that corresponds to a particular process instance.

2.3 Petri Nets

Example 1 is a Petri net representation of a simplified driver's license application process. Petri nets represent a graphical language with a formal semantics that has been used to represent, analyze, verify, and simulate dynamic behavior (Murata, 1989). Petri nets consist of places, tokens, and arcs. *Places* (drawn as circles) can contain *tokens* and are a distributed representation of state.

Each different distribution of tokens over the places of a Petri net indicate a different state. Such a state is called a *marking*. *Transitions* (drawn as rectangles) can consume and produce tokens and represent a state change. *Arcs* (drawn as arrows) connect places and transitions and represent a flow relationship. More formally, a marked Petri net is a pair $((P,T,F),s)$ where,

- P is a finite set of places,
- T is a finite set of transitions such that $P \cap T = \emptyset$, and
- $F \subseteq (P \times T) \cup (T \times P)$ is a finite set of direct arcs, and
- $s \in P \rightarrow \mathbb{N}$ is a bag over P denoting the marking of the net (van der Aalst, 1997, 1998).

Petri nets are bipartite directed graphs, that means that each arc must connect a transition to a place or a place to a transition. The transitions in a Petri net can be labeled or not. Transitions that are not labeled are called *silent transitions*. Different transitions that bear the same label are called *duplicate transitions*.

The behavior of a Petri net is determined by the *firing rule*. A transition t is *enabled* iff each input place p of t contains at least one token. When a transition is enabled it can *fire*. When a transition fires, it brings about a state change in the Petri net. In essence, it consumes one token from each input place p and produces one token in each output place p of t. To evaluate the extent to which a Petri net is capable of parsing an event sequence, transitions might be forced to fire. A transition that is not enabled, can be *forced to fire*. When a transition is forced to fire, it consumes one token from each input place that contains a token—if any—and produces one token in each output place. Petri nets are capable of representing structures such as sequences, or-splits, and-splits (parallel threads), or-joins, and-joins, loops (iterations), history-dependent or-splits, and duplicate activities that are typical for organizational processes.

3. Artificial Negative Events

Event logs generally contain sequences of positive events only. To make a tradeoff between overly general or overly specific process models, learners make additional assumptions about the given event sequences. Such assumptions are part of the inductive bias of a learner. Process discovery algorithms generally include the assumption that event logs portray the complete behavior of the underlying process and implicitly use this completeness assumption to make a tradeoff between overly general and overly specific process models. Our technique makes this completeness assumption explicit by inducing artificial negative information from the event log in a configurable way.

For processes that contain a lot of recurrent and concurrent behavior, the completeness assumption can become problematic. For example, a process containing five parallel activities (ten parallel pairs) that are placed in a loop, has $\sum_{i=1}^{n}(5!)^i$ different possible execution sequences (n being the maximum number of allowed loops).

The more recurrent behavior a process has, the more different kinds of event sequences a process can produce. This makes a given event log less likely to contain all possible behavior. The problem of recurrent behavior is addressed by restricting the window size (parameter: *windowSize*). Window size is the number of events in the subsequence one hopes to detect at least once in the sequence database. The larger the window size, the less probable that a similar subsequence is contained by the other sequences in the event log. A limited window size can be advantageous in the presence

of loops (recurrent behavior) in the underlying process. Limiting the window size to a smaller subsequence of the event log, makes the completeness assumption less strict. An unlimited window size ($windowSize = -1$) results in the most strict completeness assumption.

The problem of concurrent behavior is addressed by exploiting some available parallelism information, discovered by induction or provided as prior knowledge by a domain expert. Given a subsequence and parallelism information, all parallel variants of the subsequence can be calculated. Taking into account the parallel variants of a subsequence makes the completeness assumption less strict.The function $AllParallelVariants(\tau)$ returns the set of all parallel variants that can be obtained by permuting the activities in each sub-sequence of τ while preserving potential dependency relationships among non-parallel activities.

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 induced for this position. Algorithm 1 gives an overview of the negative event induction and is discussed in the next paragraphs. In a first step, the event log is made more compact, by grouping process instances $\sigma \in L$ that have identical sequences into grouped process instances $\tau \in M$ (line 1). By grouping similar process instances, searching for similar behavior in the event log can be performed more efficiently.

In the next step, all negative events are induced for each grouped process instance (lines 2–12). Making a completeness assumption about an event log boils down to assuming that behavior that does not occur in the event log, should not occur in the process model to be learned. Negative examples can be introduced in grouped process instances τ_i by checking at any given positive event $x_k \in \tau_i$ at position $k = Position(x_k, \tau_i)$ whether another event of interest z_k of activity type $b \in A \backslash \{AT(x_k)\}$ also could occur. For each event $x_k \in \tau_i$, it is tested whether there exists a similar sequence $\tau_j^{\parallel} \in AllParallelVariants(\tau_j) : \tau_j \neq \tau_i$ in the event log in which at that point a state transition y_k has taken place that is similar to z_k (line 6). If such a state transition does not occur in any other sequence, such behavior is not present in the event log L. This means under the completeness assumption that the state transition cannot occur. Consequently, z_k can be added as a negative event at this point k in the event sequence τ_i (lines 7–8). On the other hand, if a similar sequence is found with this behavior, no negative event is generated.

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 τ contains negative events at position k, then the ungrouped sequence σ contains corresponding negative events at position k. At each position, a large number of negative events can generally be generated. To avoid an imbalance in the proportion of negative versus positive events the addition of negative events can be manipulated with a negative event injection probability π (line 13). π is a parameter that influences the probability that a corresponding negative event is recorded in an ungrouped trace σ. The smaller π, the less negative events are generated at each position in the ungrouped event sequences. A value of $\pi = 1.0$ means that every induced negative event for a grouped sequence is included in every similar, corresponding, non-grouped sequence. A value of $\pi = 0$ will result in no negative events being induced for any of the corresponding, non-grouped sequences.

Example 2 illustrates how in an event log of two (grouped) sequences τ_1 and τ_2 artificial negative events can be generated. The event sequences originate from a simplified driver's license process, depicted at the bottom. Given the parallelism information, $Parallel(e, f)$, the event sequences each have two parallel variants. When generating negative events into event sequence τ_1, it is examined whether instead of the first event b the events c, d, e, f, g, h, or i could also have occurred at the

Algorithm 1 Generating artificial negative events

1: Group similar sequences $\sigma \in L$ into $\tau \in M$
2: **for all** $\tau_i \in M$ **do**
3: **for all** $x_k \in \tau_i$ **do**
4: $k = Position(x_k, \tau_i)$
5: **for all** $b \in A \backslash \{AT(x_k)\}$ **do**
6: **if** $\nexists \tau_j^{\parallel} \in AllParallelVariants(\tau_j) : \forall \tau_j \in M \wedge \tau_j \neq \tau_i \wedge$ $\forall y_l \in \tau_j^{\parallel}, Position(y_l, \tau_j^{\parallel}) = l = Position(x_l, \tau_j), k - windowSize < l < k : AT(y_l) = AT(x_l) \wedge$ $y_k \in \tau_j^{\parallel}, Position(y_k, \tau_j^{\parallel}) = k, AT(y_k) = b$ **then**
7: z_k = event with $AT(z_k) = b, ET(z_k) = completeRejected$
8: recordNegativeEvent(z_k, k, τ_i)
9: **end if**
10: **end for**
11: **end for**
12: **end for**
13: Induce negative events in the non-grouped sequences $\sigma \in L$ according to an injection frequency π

first position. Because there is no similar sequence $\tau_j^{\parallel}$ in which $c, d, e, f, g, h,$ or i occur at this position, it can be concluded that they are negative events. Consequently $\neg c, \neg d, \neg e, \neg f, \neg g, \neg h,$ and $\neg i$ are added as negative events at this position. Other artificial negative events are generated in a similar fashion. Notice that history-dependent processes generally will require a larger window size to correctly detect all non-local dependencies. In Example 2, an unlimited window size is used. Should the window size be limited to 1, for instance, then it would no longer be possible to take into account the non-local dependency between the activity pairs *c*–*g* and *d*–*h*. In the experiments at the end of this paper, an unlimited window size has been used (parameter value -1).

4. Process Discovery

Having an algorithm to artificially generate negative events, it becomes possible to represent process discovery as a binary classification problem, that learns the conditions that discriminate between the occurrence of an activity (a positive event), or the non-occurrence of an activity (a negative event). Algorithm 2 outlines four major steps in the AGNEs process discovery procedure. These four steps are addressed this section.

4.1 Step 1: Induce Parallelism and Locality Information

The starting point is the gathering of information about local dependency ($Local(a,b)$), and parallelism relationships ($Parallel(a,b)$, or $Serial(a,b)$) that exist between pairs of activities $a, b \in A$ in an event log L. Locality information is used in the language bias of the classification learner, to constrain the hypothesis space to locally discriminating preconditions when required. Without locality information, the classification learner is likely also to come up with non-local dependencies

Example 2 *(Continued from example 1.) DriversLicense—Generating artificial negative events for a simplified event log with two abbreviated event sequences.*

τ_1	b	c	e	f	g	i
τ_2	b	d	f	e	h	i

$Parallel(e, f)$.

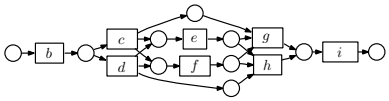
(a) given event log and background knowledge

$\tau_1^{\parallel}$	b	c	f	e	g	i
$\tau_1^{\parallel}$	b	c	e	f	g	i
$\tau_2^{\parallel}$	b	d	f	e	h	i
$\tau_2^{\parallel}$	b	d	e	f	h	i

(b) the derived parallel variants

τ_1	b	c	e	f	g	i
	$\neg c$	$\neg b$	$\neg b$	$\neg b$	$\neg b$	$\neg b$
	$\neg d$	$\neg e$	$\neg c$	$\neg c$	$\neg c$	$\neg c$
	$\neg e$	$\neg f$	$\neg d$	$\neg d$	$\neg d$	$\neg d$
	$\neg f$	$\neg g$	$\neg g$	$\neg g$	$\neg e$	$\neg e$
	$\neg g$	$\neg h$	$\neg h$	$\neg h$	$\neg f$	$\neg f$
	$\neg h$	$\neg i$	$\neg i$	$\neg i$	$\neg h$	$\neg g$
	$\neg i$				$\neg i$	$\neg h$
τ_2	b	d	f	e	h	i
	$\neg c$	$\neg b$	$\neg b$	$\neg b$	$\neg b$	$\neg b$
	$\neg d$	$\neg e$	$\neg c$	$\neg c$	$\neg c$	$\neg c$
	$\neg e$	$\neg f$	$\neg d$	$\neg d$	$\neg d$	$\neg d$
	$\neg f$	$\neg g$	$\neg g$	$\neg g$	$\neg e$	$\neg e$
	$\neg g$	$\neg h$	$\neg h$	$\neg h$	$\neg f$	$\neg f$
	$\neg h$	$\neg i$	$\neg i$	$\neg i$	$\neg g$	$\neg g$
	$\neg i$				$\neg i$	$\neg h$

(c) the induced negative events



(d) the underlying Petri net

Algorithm 2 Process discovery by AGNEs: overview

1: step 1: induce parallelism and locality from frequent temporal constraints
2: step 2: generate artificial negative events
3: step 3: learn the preconditions of each activity type
4: step 4: transform the preconditions into a Petri net

that cannot easily be transformed into a graphical model. Parallelism information is used to prevent the construction of sequential models, where in reality concurrency is expected.

This information is either gathered by the analysis of frequent temporal constraints that hold in the event log or by means of user-specified prior knowledge. Frequent temporal constraints are temporal constraints that hold in a sufficient number of sequences σ within an event log L. Goedertier (2008), defines a number of temporal constraints and shows how local dependency and parallelism information can be derived from it.

Additionally, the end-user can also specify locality and parallelism information in terms of prior knowledge. In particular, the following predicates can be used: $PriorParallel(a,b)$, $PriorSerial(a,b)$, $PriorLocal(a,b)$, and $PriorNonLocal(a,b)$. Information specified using these predicates defeats any inference about locality or parallelism made on the basis of the information gathered from frequent temporal constraints.

4.2 Step 2: Generate Artificial Negative Events

A second step in the process discovery technique is the induction of artificial negative events, as described in Section 3. The inferred information about parallel activity pairs can be used for parallel

variant calculation. Furthermore, the induction of negative events is dependent on the window size *windowSize*, and negative event injection probability π parameters.

4.3 Step 3: Learn the Preconditions of each Activity Type

It is now possible to represent process discovery as a multiple, first-order classification learning problem that learns the preconditions that discriminate between the occurrence of either a positive or a negative completion event for each activity type. In our experiments, we make use of TILDE, an existing multi-relational classification learner, to perform the actual classification learning.

The motivation for representing process discovery as a classification problem is that it allows using classification learning and evaluation techniques that are well-known in the machine learning community. Furthermore, classification learners have the potential to deal with so-called **duplicate tasks**. Duplicate tasks refer to the reoccurrence of identically labeled transitions, homonyms, in different contexts within the event logs. Classification learning can detect the different execution contexts for these transitions and derive different preconditions for them. Transforming these preconditions into graphs will eventually result in duplicate, homonymic activities in the graph model that correspond to the different usage contexts. Techniques for process discovery, such as heuristics miner and genetic miner, which have causal matrices as internal representation (Weijters et al., 2006; Alves de Medeiros et al., 2007), are unable to discover duplicate activities. Alves de Medeiros (2006), however, presents a non-trivial extension of the genetic miner that includes duplicate tasks.

The motivation for using a multi-relational, first-order representation is that first-order learning allows relating the occurrence of an event to the occurrence of any other event. This enables the detection of patterns that involve non-local, historic, dependencies between events. Alternatively, the history of each event could in part be represented as extra propositions, for instance by including all preceding positive events as extra columns in the event log. This propositional representation would have many difficulties.

Being able to detect non-local, historic patterns in an event log can also produce counter-intuitive results. A non-local relationship might have more discriminating power than a local one and can therefore be preferred by a learner. Unfortunately, an excess of non-local patterns makes it more difficult to generate a graphical model, a Petri net, containing local connections. Because TILDE allows specifying a language bias with dynamic refinement (Blockeel and De Raedt, 1998), it becomes possible to constrain the hypothesis space to locally discriminating patterns whenever necessary. One technique is the dynamic generation of constants, that can be used to constrain the combinations of activity type constants that are to be considered for a particular language bias construct. Additionally, it is also possible to constrain the occurrence of particular literals in a hypothesis $\mathcal{H}$, given the presence or absence of other literals that are already part of the hypothesis.

The classification task of TILDE is to predict whether for a given activity type $a \in A$, at a given time point $t \in \mathbb{N}$ in a given sequence $\sigma \in L$, a positive or a negative event takes place. In the case of a positive event, the target predicate evaluates to $\mathit{Class}(a, \sigma, t, \mathit{completed})$. In the case of a negative event, the target predicate evaluates to $\mathit{Class}(a, \sigma, t, \mathit{completeRejected})$. In the language bias, the target activity, indicated by a, will be used for dynamically constraining the combinations of activity type constants generated.

The primary objective of AGNEs being the construction of a graphical model from an event log, the language bias consists of a logical predicate that can represent the conditions under which a Petri net place contains a token. This predicate is called the "no-sequel" predicate, $\mathit{NS}(a_1, a, \sigma, t)$. Let

$a_1, a \in A$ be activity types, $\sigma \in L$ the sequence of a process instance, and t the time of observation. The *NS* predicate is defined as follows:

$$\forall a_1, a \in A, t \in \mathbb{N} : \exists x \in \sigma : AT(x) = a_1 \ \wedge \ Time(x) < t$$
$$\wedge \nexists y \in \sigma : AT(y) = a \ \wedge \ Time(x) < Time(y) < t \quad \Rightarrow NS(a_1, a, \sigma, t).$$

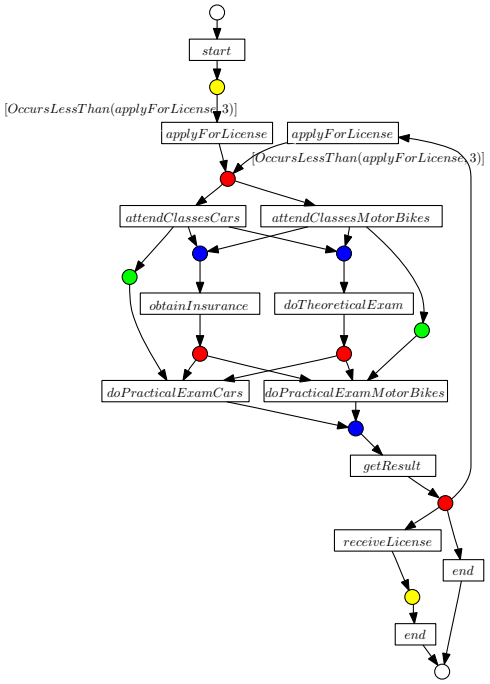
In the remainder of this text, the arguments σ and t will be implicitly assumed and therefore left out. The predicate $NS(a_1, a)$ evaluates to true when at the time of observation, an activity a_1 has completed, but has not (yet) been followed by an activity a. In combination with conjunction ($\wedge$), disjunction ($\vee$) and negation-as-failure ($\sim$), the $NS(a_1, a)$ predicate makes it possible to learn fragments of Petri nets using a multi-relational classification learner.

Example 3 shows how the preconditions in the Petri net can be represented as conjunctions and disjunctions of *NS* atoms. This representation accounts for the different contexts in which the activities *applyForLicense* and *end* can take place and derives different preconditions for these activities. In addition, it can represent the non-local, non-free, history-dependent choice construct between the activities *attendClassesCars–doPracticalExamCars* and *attendClassesMotorBikes–doPracticalExamMotorBikes* and the maximum recurrence of the activity *applyForLicense*. The Petri net fragments included in the language bias of AGNEs are enumerated in Figure 1 and are briefly discussed in the remainder of this section.

Example 3 *(Continued from example 2) DriversLicensel—A Petri net in terms of NS preconditions*

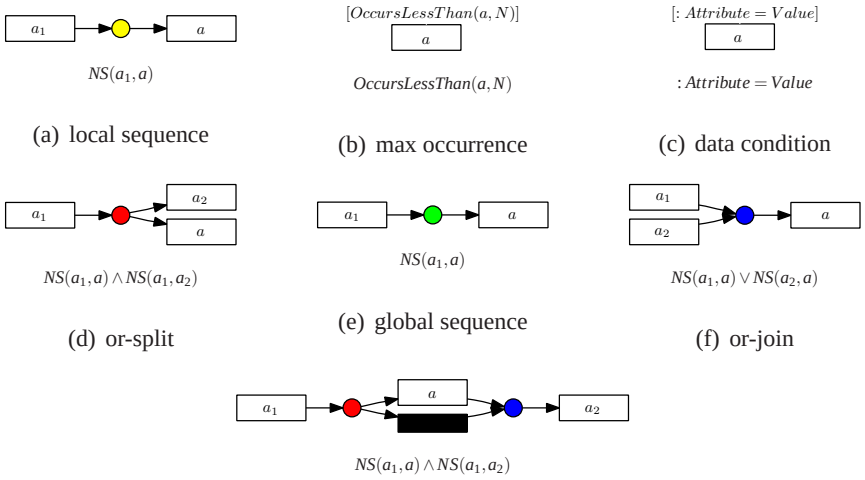
	activity	***precondition***
a	*start*	$true$
b	*applyForLicense*	$NS(a,b)$
b	*applyForLicense*	$(NS(i,b) \wedge NS(i,j) \wedge NS(i,k))$ $\wedge$ $OccursLessThan(b,3)$
c	*attendClassesCars*	$NS(b,c) \wedge NS(b,d)$
d	*attendClassesMotorBikes*	$NS(b,c) \wedge NS(b,d)$
e	*obtainInsurance*	$NS(c,e) \vee NS(d,e)$
f	*doTheoreticalExam*	$NS(c,f) \vee NS(d,f)$
g	*doPracticalExamCars*	$(NS(f,g) \wedge NS(f,h)) \wedge$ $(NS(e,g) \wedge NS(e,h)) \wedge NS(c,g)$
h	*doPracticalExamMtrBikes*	$(NS(f,g) \wedge NS(f,h)) \wedge$ $(NS(e,g) \wedge NS(e,h)) \wedge NS(d,h)$
i	*getResult*	$NS(g,i) \vee NS(h,i)$
j	*receiveLicense*	$NS(i,b) \wedge NS(i,j) \wedge NS(i,k)$
k	*end*	$NS(j,k)$
k	*end*	$NS(i,b) \wedge NS(i,j) \wedge NS(i,k)$

(a) DriversLicensel—activity preconditions



(b) DriversLicensel—Petri net

Figure 1(a) shows a graphical representation of the local sequence predicate that is part of the language bias. A local sequence of two transitions labeled $a_1, a \in A$ in a Petri net can be represented by $NS(a_1, a)$. In the language bias, the following constraints apply:



(a) local sequence (b) max occurrence (c) data condition

(d) or-split (e) global sequence (f) or-join

(g) skip sequence

Figure 1: Petri net patterns in the language bias

sequence	$NS(a_1, a)$	
constraints:	$Local(a_1, a)$.	(1)
	$\nexists l \in \mathcal{H} : l = [NS(\_, a_1)]$.	(2)

The conversion from *NS*-based preconditions into Petri nets requires the conditions to refer to local, immediately preceding events as much as possible. Therefore, constraint (1) requires to restrict the generation of constants a_1 to constants that are local to a. In constraint (1) we do not require a_1 and a to be different activity types. This is sufficient for the discovery of length-one loops. Graphically, a conjunction of *NS* constructs can be considered to be the logical counterpart of a single Petri net place. Besides the case of length-one loops—that can be expressed with a single *NS* construct—there is no reason for a Petri net place to contain both an input and an output arc that is connected to the same transition. Constraint (2) has been put in place to keep a multi-relational learner from constructing such useless hypotheses. The constraint stipulates that the construct $NS(a_1, a)$ can only be added to the current hypothesis $\mathcal{H}$, if $\mathcal{H}$ does not already contain a logical condition that boils down to an output arc towards a_1.

The same $NS(a_1, a)$ construct, with different constraints, can be used to keep track of a global sequence (see Figure 1(e)). Global sequences are used to represent non-local, non-free choice constructs. We require TILDE only to consider global sequence between activity types for which both a precedence and a response relationship has been detected from the event log, this is expressed as a language bias constraint. Because we assume that any transition must be locally connected, that is connected to other transitions to which it is local in the event log, we require as an additional constraint that the global sequence construct must not be added first to any hypothesis. The other Petri net based language constructs depicted in Figure 1 have similar definitions and are described in detail by Goedertier (2008).

The language bias of TILDE is limited to conjunctions and disjunctions of *NS* constructs of length two and three. Or-splits and or-joins that involve more activity types are obtained by grouping conjunctions and disjunctions of *NS* constructs into larger conjunctions and disjunctions in step 4. However, this limitation in length sometimes leads to TILDE make inadequate refinements. Solving this language bias issue, requires constructing a proprietary ILP classification algorithm that during each refinement step allows considering conjunctions of *NS* constructs of variable lengths. We leave this improvements to future work.

4.4 Step 4: Transform the Preconditions into a Petri Net

In the previous step, TILDE has learned preconditions for each activity type independently. In a Petri net, the preconditions for each activity type are nonetheless interrelated. Therefore, the logic programs of activity preconditions are submitted to several rule-level pruning steps, to make sure that they do not produce redundant duplicate places in the Petri net to be constructed. These pruning steps occur among the conditions within a single precondition, within a set of preconditions to the same activity type and among preconditions of activity types that pertain to the same or-split. Given a pruned rule set of preconditions for each activity type, the construction of a Petri net is fairly straightforward. Each induced precondition corresponds to a different transition in a Petri net to be constructed. Because AGNEs may produce several preconditions for an activity type, the constructed Petri net may contain duplicate transitions. Algorithm 3 provides an overview of these procedures. In the remainder of this section, these different pruning steps will be discussed and illustrated for a sample of rules from the DriversLicensel example.

The logic programs constructed by TILDE contain rules that classify the occurrence of either a positive or a negative event. By construction, the language bias of AGNEs is such that TILDE will never consider a negation of an *NS* construct to be explanatory for the occurrence of a positive event. Therefore, TILDE will never construct a tree in which a right leaf predicts a positive event. The latter entails that, in equivalent the logic program, the rules with a $class(a,\sigma,t,comleted)$ rule head can be taken from the logic programs without loss of information (line 1). Example 4 depicts the preconditions that are induced by TILDE for the apply for license and end activities in the DriversLicensel example.

In a second step, a number of intra-rule pruning operations take place (lines 2–6). The top-down refinement of hypotheses, can result in the derivation of logically redundant conditions. These logical redundancies are removed for each rule (line 3). Each rule in the logic program consists of conjunctions of groups of *NS* constructs. A conjunction of a pair of or-split constructs that originate from the same activity a_1 can be combined into a larger or-split (line 4). Likewise, a conjunction of a pair of or-joins can be combined into a larger or-join (line 5). Example 4 illustrates intra-rule pruning for the DriversLicensel example.

In a third step, a number of inter-rule pruning operations are performed for the preconditions that pertain to each activity type. In particular, a precondition that subsumes another precondition for the same activity type, is redundant and thus removed from consideration (lines 10–12). Furthermore, it is examined whether a more specific or-join can be constructed out of the groups of *NS* constructs within the different preconditions of the same rule (lines 13–15). Finally, all rules are examined to find the most specific or-split condition from the preconditions extracted by TILDE (lines 18–26). Example 4 illustrates this pruning for the DriversLicensel example.

Example 4 *(Continued from example 3.) DriversLicensel—pruning*

rules induced by TILDE	:
b apply for license	$(NS(i,b) \vee NS(a,b)) \wedge (NS(i,b) \wedge NS(i,j))$.
b apply for license	$(NS(i,b) \vee NS(a,b))$.
k end	$(NS(j,k))$.
k end	$(NS(i,b) \wedge NS(i,k))$.
intra-rule pruning:	
b apply for license	$(NS(i,b) \wedge NS(i,j))$. *(line 3)*
b apply for license	$(NS(i,b) \vee NS(a,b))$.
k end	$(NS(j,k))$.
k end	$(NS(i,b) \wedge NS(i,k))$.
inter-rule pruning, most specific or split:	
b apply for license	$(NS(i,b) \wedge NS(i,j) \wedge NS(i,k))$. *(lines 18–26)*
b apply for license	$(NS(a,b))$. *(lines 10–12)*
k end	$(NS(j,k))$.
k end	$(NS(i,b) \wedge NS(i,j) \wedge NS(i,k))$. *(lines 18–26)*

5. Evaluation Metrics

Discovered process models preferably allow the behavior in the event log (recall) but no other, unobserved, random behavior (specificity). Having formulated process discovery as a binary classification problem on event logs supplemented with artificial negative events, it becomes possible to use the true positive and true negative rate from classification learning theory to quantify the recall and specificity of a process model:

- **true positive rate** TP_{rate} or **recall**: the percentage of correctly classified positive events in the event log. This probability can be estimated as follows: $TP_{rate} = \frac{TP}{TP+FN}$, where TP is the amount of correctly classified positive events and FN is the amount of incorrectly classified positive events.

- **true negative rate** TN_{rate} or **specificity**: the percentage of correctly classified negative events in the event log. This probability can be estimated as follows: $TN_{rate} = \frac{TN}{TN+FP}$, where TN is the amount of correctly classified negative events and FP is the amount of incorrectly classified negative events.

Accuracy is the sum of the true positive and true negative rate, weighted by the respective class distributions. The fact that accuracy is relative to the underlying class distributions can lead to unintuitive interpretations. Moreover, in process discovery, these class distributions can be quite different and have no particular meaning. In order to make abstraction of the proportion of negative and positive events, we propose, from a practical viewpoint to attach equal importance to both recall and specificity: $acc = \frac{1}{2}recall + \frac{1}{2}specificity$. According to this definition, the accuracy of a majority-class predictor is $\frac{1}{2}$. Flower models, such as the one in Example 1(b), are an example of such a majority-class predictors. Because a flower model represents random behavior, it has a perfect recall of the all behavior in the event log but it also has much *additional* behavior compared to the event log. Because of the latter fact, the flower model has zero specificity, and an accuracy of $\frac{1}{2}$. Any useful process model should have an accuracy higher than $\frac{1}{2}$.

Algorithm 3 Rule-level pruning

$R^+ = \{r \in R \mid r \text{ has rule head } class(a, \sigma, t, comleted)\ \}$.
// intra-rule pruning:
for all $r \in R^+$ **do**
  reduce r according to $(NS_1 \vee NS_2) \wedge NS_1 \equiv NS_1$.
  group or-splits: $NS(a_1,a) \wedge NS(a_1,a_2) \in r$ and $NS(a_1,a) \wedge NS(a_1,a_3) \in r$ into $NS(a_1,a) \wedge NS(a_1,a_2) \wedge NS(a_1,a_3)$.
  group or-joins: $NS(a_1,a) \vee NS(a_2,a) \in r$ and $NS(a_3,a) \vee NS(a_4,a) \in r$ into $NS(a_1,a) \vee NS(a_2,a) \vee NS(a_3,a) \vee NS(a_4,a)$.
end for
//inter-rule pruning:
for all $a \in A$ **do**
  $R_a^+ = \{r \in R^+ \mid r \text{ is a precondition of } a\}$
  for all $r \in R_a^+$ **do**
    if $\exists s \in R_a^+ : s$ is more specific than r **then**
      remove r.
    end if
    if $\exists s \in R_a^+ : s$ combined with r lead to a more specific or-join than r **then**
      replace the or-join in r with the more specific or-join.
    end if
  end for
end for
//keep the most specific or-split:
for all $a \in A$ **do**
  $R_a^{split} = \{r \in R^+ \mid r \text{ contains an or-split going out } a\ \}$.
  s = the most specific or-split by combining the or-splits going out a in R_a^{split}.
  for all $r \in R_a^{split}$ **do**
    if s is more specific than r **then**
      replace the or-split in r with the or-split in s.
    end if
  end for
end for

The metrics that are introduced in this section will be used to evaluate the performance of AGNEs in the following sections. They have been defined and implemented for Petri nets. However, they can also be applied to other generative models.

5.1 Existing Metrics

Weijters et al. (2006) define a metric that has a somewhat similar interpretation as TP_{rate}: the parsing measure PM. The measure is defined as follows:

- **parsing measure** PM: the number of sequences in the event log that are correctly parsed by the process model, divided by the total number of sequences in the event log. For efficiency, the similar sequences in the event log are grouped. Let k represent the number of grouped

sequences, n_i the number of process instances in a grouped sequence i, c_i a variable that is equal to 1 if grouped sequence i can be parsed correctly, and 0 if grouped sequence i cannot be parsed correctly. The parsing measure can be defined as follows Weijters et al. (2006):

$$PM = \frac{\sum_{i=1}^{k} n_i c_i}{\sum_{i=1}^{k} n_i}.$$

PM is a coarse-grained metric. A single missing arc in a Petri net can result in parsing failure for all sequences. A process model with a single point of failure is generally better than a process model with more points of failure. This is not quantified by the parsing measure *PM*.

Rozinat and van der Aalst (2008) define two metrics that have a somewhat similar interpretation as TP_{rate} and TN_{rate}: the fitness metric f and the advanced behavioral appropriateness metric $a_B^{'}$:

- **fitness** f: Fitness is a metric that is obtained by trying whether each (grouped) sequence in the event log can be reproduced by the generative model. This procedure is called *sequence replay*. The metric assumes the generative model to be a Petri net. At the start of the sequence replay, f has an initial value of one. During replay, the transitions in the Petri net will produce and consume tokens to reflect the state transitions. However, the proportion of tokens that must additionally be created in the marked Petri net, so as to *force* a transition to fire, is subtracted from this initial value. Likewise, the fitness measure f punishes for extra behavior by subtracting the proportion of remaining tokens relative to the total number of produced tokens from this initial value. Let k represent the number of grouped sequences, n_i the number of process instances, c_i the number of tokens consumed, m_i the number of missing tokens, p_i the number of produced tokens, and r_i the number of remaining tokens for each grouped sequence i $(1 \leq i \leq k)$. The fitness metric can be defined as follows (Rozinat and van der Aalst, 2008):

$$f = \frac{1}{2}\left(1 - \frac{\sum_{i=1}^{k} n_i m_i}{\sum_{i=1}^{k} n_i c_i}\right) + \frac{1}{2}\left(1 - \frac{\sum_{i=1}^{k} n_i r_i}{\sum_{i=1}^{k} n_i p_i}\right).$$

- **behavioral appropriateness** $a_B^{'}$: Behavioral appropriateness is a metric that is obtained by an exploration of the state space of a Petri net and by comparing the different types of *follows* and *precedes* relationships that occur in the state space with the different types of *follows* and *precedes* relationships that occur in the event log. The metric is defined as the proportion of number of *follows* and *precedes* relationships that the Petri net has in common with the event log vis-à-vis the number of relationships allowed by the Petri net. Let S_F^m be the S_F relation and S_P^m be the S_P relation for the process model, and S_F^l the S_F relation and S_P^l the S_P relation for the event log. The advanced behavioral appropriateness metric $a_B^{'}$ is defined as follows (Rozinat and van der Aalst, 2008):

$$a_B^{'} = \left(\frac{|S_F^l \cap S_F^m|}{2.|S_F^m|} + \frac{|S_P^l \cap S_P^m|}{2.|S_P^m|}\right).$$

Rozinat and van der Aalst (2008) also report a solution to two non-trivial problems that are encountered when replaying Petri nets with silent steps and duplicate activities. In the presence of silent steps (or invisible tasks) it is non-trivial to determine whether there exist a suitable firing sequence

of invisible tasks such that the right activities become enabled for the Petri net to optimally replay a given sequence of events. Likewise, in the presence of multiple enabled duplicate activities, it is a non-trivial problem of determining the optimal firing, as the firing of one duplicate activity affects the ability of the Petri net to replay the remaining events in a given sequence of events. Rozinat and van der Aalst (2008) present two local approaches that are based on heuristics involving the next activity event in the given sequence of events.

Fitness f and behavioral appropriateness $a_B^{'}$ are particularly useful measures to evaluate the performance of a discovered process model. Moreover, these metrics have been implemented in the ProM framework. However, the interpretation of the fitness measure requires some attention: although it accounts for recall as it punishes for the number of missing tokens that had to be created, it also punishes for the number of tokens that remain in a Petri net after log replay. The latter can be considered *extra* behavior. Therefore, the fitness metric f also has a specificity semantics attached to it. Furthermore, it is to be noted that the behavioral appropriateness $a_B^{'}$ metric is not guaranteed to account for all non-local behavior in the event log (for instance, a non-local, non-free, history-dependent choice construct that is part of a loop will not be detected by the measure). In addition, the $a_B^{'}$ metric requires an analysis of the state space of the process model or a more or less exhaustive simulation to consider all allowable sequences by the model.

5.2 New Metrics

The availability of an event log supplemented with artificial negative events, allows for the definition of a new specificity metric that does not require a state space analysis. Instead, specificity can be calculated by parsing the (grouped) sequences, supplemented with negative events. We therefore define:

- **behavioral recall** r_B^p: The behavioral recall r_B^p metric is obtained by parsing each grouped event sequence. The values for *TP* and *FN* are initially zero. Starting from the initial marking, each sequence is parsed. Whenever an enabled transitions fires, the value for *TP* is increased by one. Whenever a transition is not enabled, but must be forced to fire the value for *FN* is increased. As an optimization, identical sequences are only replayed once. Let k represent the number of grouped sequences, n_i the number of process instances, TP_i number of events that are correctly parsed, and FN_i the number events for which a transition was forced to fire for each grouped sequence i $(1 \leq i \leq k)$. At the end of the sequence replay, r_B^p is obtained as follows:

$$r_B^p = \left(\frac{\sum_{i=1}^{k} n_i TP_i}{\sum_{i=1}^{k} n_i TP_i + \sum_{i=1}^{k} n_i FN_i} \right).$$

  In the case of multiple enabled duplicate transitions, sequence replay fires the transition of which the succeeding transition is the next positive event (or makes a random choice). In the case of multiple enabled silent transitions, log replay fires the transition of which the succeeding transition is the next positive event (or makes a random choice). Unlike the fitness metric f, r_B^p does not punish for remaining tokens. Whenever after replaying a sequence tokens remaining tokens cause *additional behavior* by enabling particular transitions, this is punished by our behavioral specificity metric s_B^n.

- **behavioral specificity** s_B^n: The behavioral specificity s_B^n metric can be obtained during the same replay as r_B^p. The values for *TN* and *TP* are initially zero. Whenever during replay,

a negative event is encountered for which no transitions are enabled, the value for *TN* is increased by one. In contrast, whenever a negative event is encountered during sequence replay for which there is a corresponding transition enabled in the Petri net, the value for *FP* is increased by one. As an optimization, identical sequences only are to be replayed once. Let k represent the number of grouped sequences, n_i the number of process instances, TN_i number of negative events for which no transition was enabled, and FP_i the number negative events for which a transition was enabled during the replay of each grouped sequence i $(1 \leq i \leq k)$. At the end of the sequence replay, s_B^n is obtained as follows:

$$s_B^n = \left(\frac{\sum_{i=1}^{k} n_i TN_i}{\sum_{i=1}^{k} n_i TN_i + \sum_{i=1}^{k} n_i FP_i} \right).$$

Because the behavioral specificity metric s_B^n checks whether the Petri net recalls negative event, it is inherently dependent on the way in which these negative events are generated. For the moment, the negative event generation procedure is configurable by the negative event injection probability π, and whether or not it must account for the parallel variants of the given sequences of positive events. For the purpose of uniformity, negative events are generated in the test sets with π equal to 1.0 and account for parallel variants = true.

6. Experimental Evaluation

AGNEs has been implemented in Prolog. In particular, the frequent temporal constraint induction, the artificial negative event generation, the language bias constraints, and the pruning and graph construction algorithms all have been written in Prolog. As mentioned before, AGNEs makes use of TILDE, an existing multi-relational classifier (Blockeel and De Raedt, 1998), available in the ACE-ilProlog data mining system (Blockeel et al., 2002). To be able to benefit from the facilities of the ProM framework, a plugin was written that makes AGNEs accessible in ProM.[1] Figure 3 depicts a screen shot of AGNEs in ProM.

In this section the results of an experimental evaluation of AGNEs are presented. First we will discuss the properties of the event logs and the parameter settings that have been used. Then, in Section 6.1, we analyse the expressiveness of AGNEs and benchmark the ability of AGNEs to generalize from incomplete event logs. In Section 6.2, we analyze the results of a number of noise experiments with different types and levels of noise that have been carried out to test how the learning algorithm behaves in the presence of noise.

In order to evaluate and compare the performance of AGNEs, a benchmark experiment with 34 event logs has been set up. These event logs have previously been used by Alves de Medeiros (2006) and Alves de Medeiros et al. (2007) to evaluate the genetic miner algorithm. Table 1 describes the properties of the underlying artificial process models of the event logs. The number of different process instance sequences (column "$\neq$ process inst.") gives an indication of the amount of different behavior that is present in the event log. This number is to be compared with the total number of process instances in the event logs. In general, the presence of loops and parallelism exponentially increases the amount of different behavior that can be produced by a process. Therefore, the number of activity types that are pairwise parallel and the number and type of loops have been reported in Table 1. In correspondence with the naming conventions used by Alves de Medeiros, nfc stands for

1. The AGNEs plugin is available from `http://www.processintelligence.be/AGNEs.php`.

non-free-choice, l1l and l2l stands for the presence of a length-one and length-two loop respectively, and st and unst stands for structured and unstructured loops. Furthermore, the presences of special structures such as skip activities and (parallel or serial) duplicate activities have been indicated. For most of the event logs in the experiment, a reference model was available that can be assumed to represent the behavior in the event log. Columns "r_B^p reference model" and "s_B^n reference model" indicate the behavioral recall and specificity of the reference models with respect to the original event logs.

In the experiments, the performance of AGNEs is compared to the performance of four state-of-the-art process discovery algorithms: α^+ (van der Aalst et al., 2004; Alves de Medeiros et al., 2004), α^{++} (Wen et al., 2007), genetic miner (Alves de Medeiros et al., 2007) and heuristics miner (Weijters et al., 2006). Being the first large-scale, comparative benchmark study in the literature of process discovery, we have chosen to include algorithms that already have appeared as journal publications (α^+, α^{++}, and genetic miner) or that are much referenced in the literature (heuristics miner).

During all experiments, the algorithms were run with the same, standard parameter settings of their ProM 4.2 implementation, as reported in Table 2. These parameter settings coincide with the ones used to run similar experiments by the authors of the algorithms. To enable a comparison on the same terms, AGNEs was not provided with prior knowledge regarding parallelism or locality of activity types. In particular, the thresholds used to induce frequent temporal patterns have been given the following values ($t_{absence}$ = 0.9, t_{chain} = 0.08, t_{succ} = 0.8, $t_{ordering}$ = 0.08, t_{triple} = 0.10). In practice, a good threshold depends on the amount of low-frequent behavior (noise) one is willing to accept within the discovered process model. The negative event injection probability π influences the proportion of artificially generated negative events in an event log. A strong imbalance of this proportion may bias a classification learner towards a majority class prediction, without deriving any useful preconditions for a particular activity type. As a rule of thumb, it is a good idea to set this parameter value as low as possible, without the learner making a majority class prediction. In the experiments π has been given a default value of 0.08. Ex-post, AGNEs warns the user when too low a value for π has led to a majority-class prediction. TILDE's C4.5 gain metric was used as a heuristic for selecting the best branching criterion. In addition, TILDE's C4.5 post pruning method was used with a standard confidence level of 0.25. Furthermore, TILDE is forced to stop node splitting when the number of process instances in a tree node drops below 5. The same parameter settings have been used on all 34 data sets. Empirical validation has shown the parameter settings to work well across all data sets.

The AGNEs technique, has run times in between 20 seconds and 2 hours for the data sets in the experiments on a Pentium 4, 2.4 Ghz machine with 1GB internal memory. These processing times are well in excess of the processing times of α^+, α^{++} and heuristics miner. In comparison to the run times of the genetic miner algorithm, processing times are considerably shorter. Most of the time is required by TILDE to learn the preconditions for each activity type. The generation of negative events also can take up some time. As process discovery generally is not a real-time data mining application, less attention has been given to computation times.

6.1 Zero-noise, Cross Validation Experiment

To evaluate AGNEs' expressiveness and ability to generalize, a 10-fold cross-validation experiment has been set up. In the literature on process discovery, cross-validation has only been considered by

	activity types	$\neq$ process inst.	process inst.	r_B^p reference model	s_B^n reference model	$\parallel$ activity types	loops	skip	nfc	duplicate
a10skip	12	6	300	1.000	1.000	1		1		
a12	14	5	300	1.000	1.000	2				
a5	7	13	300	1.000	1.000	1	1 l1l			
a6nfc	8	3	300	1.000	1.000	1			1	
a7	9	14	300	1.000	1.000	4				
a8	10	4	300	1.000	1.000	1				
al1	9	98	300	1.000	0.996	n.a.	1 unst			
al2	13	92	300	1.000	0.992	n.a.	2 unst			
betaSimplified	13	4	300	1.000	1.000	0		1	1	2
bn1	42	4	300	1.000	1.000	0				
bn2	42	25	300	1.000	1.000	0	1 st			
bn3	42	150	300	1.000	0.999	0	2 st			
choice	12	16	300	1.000	1.000	0				
DriversLicense	9	2	300	1.000	1.000	0				
DriversLincensel	11	87	350	1.000	0.986	1	1 st	1	1	1
herbstFig3p4	12	32	300	1.000	0.999	3	1 st			
herbstFig5p19	8	6	300	1.000	1.000	1				1
herbstFig6p18	7	153	300	1.000	0.977	0	1 l1l, 1 l2l			
herbstFig6p19	5	136	300	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
herbstFig6p31	9	4	300	1.000	1.000	0				1
herbstFig6p33	10	4	300	1.000	1.000	0				1
herbstFig6p36	12	2	300	1.000	1.000	0			1	
herbstFig6p37	16	135	300	1.000	0.996	36				
herbstFig6p38	7	5	300	1.000	1.000	3				1 par.
herbstFig6p39	7	12	300	1.000	1.000	1				
herbstFig6p41	16	12	300	1.000	1.000	4				
herbstFig6p45	8	12	300	1.000	1.000	5				
l1l	6	69	300	1.000	0.988	1	2 l1l			
l1lSkip	6	269	300	1.000	0.732	0	2 l1l			
l2l	6	10	300	1.000	1.000	0	1 l2l			
l2lOptional	6	9	300	1.000	1.000	0	1 l2l			
l2lSkip	6	8	300	1.000	0.999	0	1 l2l			
parallel5	10	109	300	1.000	1.000	10				
repair2	8	48	1000	0.998	0.995	2	1 unst			

Table 1: Event log properties

Goedertier et al. (2008) and Rozinat et al. (2007). The reason for the absence of cross-validation experiments, is that process discovery is an inherently *descriptive* learning task rather than a *predictive* one. The primary intent of process discovery is to produce a model that accurately describes the event log at hand. Nonetheless, it is interesting to test the *predictive* ability of process discovery algorithms in an experimental setting. To apply cross-validation, a randomization routine has been

Algorithm (Ref.)	Parameter settings	
α^+ (Alves de Medeiros et al., 2004)	derive succession from partial order information = true enforce causal dependencies within events of the same activity = false enforce parallelism by overlapping events = false	
α^{++} (Wen et al., 2007)	(no settings)	
heuristics miner Weijters et al. (2006)	relative to best threshold = 0.05 positive observations = 10 dependency threshold = 0.9 length-one-loops threshold = 0.9 length-two-loops threshold = 0.9 long-distance threshold = 0.9 dependency divisor = 1 and threshold = 0.1 use all-activities-connected heuristic = true use long-distance dependency heuristic = false	
genetic miner (Alves de Medeiros et al., 2007)	population size = 100 max number generations = 1000 initial population type = possible duplicates power value = 1 elitism rate = 0.2 selection type = tournament 5 extra behavior punishment with $\kappa = 0.025$ enhanced crossover type with crossover probability = 0.8 enhanced mutation type with mutation probability = 0.2	
AGNEs	prior knowledge	none
	temporal constraints	$t_{absence} = 0.9$, $t_{chain} = 0.08$, $t_{succ} = 0.8$, $t_{ordering} = 0.08$, $t_{triple} = 0.1$
	negative event generation	injection probability $\pi = 0.08$ calculate parallel variants = true include global sequences = true
	language bias: TILDE	include occurrence count = false data conditions = none splitting heuristic: gain minimal cases in tree nodes = 5 C4.5 pruning with confidence level = 0.25
	graph construction	$t_{connect} = 0.4$

Table 2: Parameter settings

written in SWI-Prolog that groups similar sequences, randomly partitions the grouped event log in $n = 10$ uniform subgroups, and produces n pairs of training and test event logs. Training event logs are used for the purpose of process discovery. Test event logs are used for evaluation, this is for calculating the specificity and recall metrics. For the purpose of this experiment, no noise was added to the event logs.

Specificity metrics must be calculated based on the combination of training and test event logs, the entire event log. Although this might seem unintuitive, specificity and specificity metrics make a completeness assumption as well, as they account for the amount of *extra* behavior in a process model vis-à-vis the event log (Rozinat and van der Aalst, 2008). To correctly evaluate the proposed learning technique, it is important that the negative events in the test set accurately indicate the state transitions that are not present in the event log. For this reason, the negative events in the test log are created with information from the entire event log. Should the negative event generation be based on training set instances only, it is possible that additional, erroneous negative events are injected because it is possible that some behavior is not present in the test set. In short, the experiment applies the above-described partitioning, after having generated negative events for each grouped process instance. Intended to be used by the evaluation metric, the negative events have been generated with

an injection probability π equal to 1, an infinite window size, and by considering parallel variants. Evidently, the thus generated negative events were not retained in the training set. For training purposes, negative events have been calculated based on the information in the training set only. For the same reasons, the behavioral appropriateness metric a'_B has also been calculated based on the whole of training and test set data.

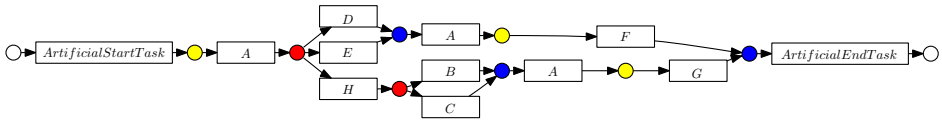
In the experiment, only 19 out of the 34 event logs were retained, as the other event logs have less than 10 different sequences. Table 3 shows the aggregated, average results of the 10-fold cross validation experiment over 190 event logs. The **best** average performance over the 34 event logs is underlined and denoted in bold face for each metric. We then use a paired t-test to test the significance of the performance differences (Van Gestel et al., 2004). Performances that are **not significantly different at the 5% level** from the top-ranking performance with respect to a one-tailed paired t-test are tabulated in bold face. Statistically *significant underperformances at the 1% level* are emphasized in italics. Performances significantly different at the 5% level but not at the 1% level are reported in normal font. For the *PM* measure, no paired t-tests could be performed, because the metric could not be calculated on some of the process models discovered by α^+ and α^{++}. The latter is the case when the discovered process models have disconnected elements.

From the results for the parsing measure *PM*, the fitness measure f, and behavioral recall measures r_B^p, it can be concluded that genetic miner scores slightly better on the recall requirement. Moreover, the behavioral specificity metric s_B^n shows genetic miner and heuristics miner to produce slightly more specific models.

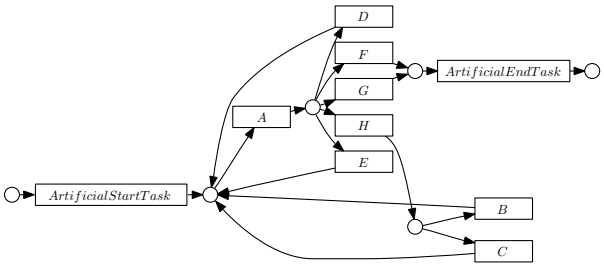
		PM	f	a'_B	*acc*	r_B^p	s_B^n	acc_B^{pn}
zero_noise	α^+	0.72	*0.96*	**0.96**	**0.96**	*0.97*	*0.83*	*0.90*
	α^{++}	0.77	*0.97*	*0.81*	*0.88*	0.97	*0.90*	0.93
	AGNEs	0.80	0.98	*0.81*	*0.89*	**0.98**	*0.91*	0.94
	genetic	0.83	**0.99**	*0.84*	*0.91*	**0.98**	**0.93**	**0.95**
	heuristics	0.79	*0.97*	*0.85*	*0.91*	0.97	**0.93**	**0.95**

Table 3: 10-fold cross validation experiment - aggregated results

From the cross-validation experiment, we conclude that AGNEs portrays similar generalization behavior to other process discovery algorithms. The reason that it is not sensitive to incomplete event logs can be attributed to the following. Given an incomplete event log, AGNEs is likely to generate a proportion of incorrect negative events. However, this proportion of negative events is relatively small, as the negative event injection parameter π is not required to be excessively large. More importantly, the coarse-grained language bias that combines *NS* constructs into larger structures, prevents TILDE from overfitting the incomplete event log and allows it to generalize, to some extent, beyond the observed behavior. The additional incorporation of process knowledge expressed by a domain expert would only add to this benefit. Finally, the negative event injection procedure takes into account parallelism and window size. Concurrent and recurrent behavior are the root causes of incomplete event logs. The ability to include information about parallel variants and window size, gives our learning technique a configurable inductive bias, with different strategies to account for incompleteness.



(a) AGNEs result



(b) heuristics miner result

Figure 2: herbstFig6p33: AGNEs detects the duplicate activity *A*

The metrics r_B^p and s_B^n do not indicate a large or significant difference for the performance of α^{++}, AGNEs, genetic miner, and heuristics miner. Only by looking at the individual process models, the expressiveness of AGNEs with respect to the detection of non-local, non-free choice constructs or the discovery of duplicate tasks becomes apparent. Example 3, discussed in Section 4.3, shows how AGNEs is capable of detecting non-local, non-free choice constructs, even within the loop of the DriversLicensel reference problem. AGNEs is also particularly suited for the detection of duplicate activities. In the herbstFig6p33 event log, the activity *A* occurs in three different contexts and AGNEs draws three different, identically labeled transitions correspondingly. Figure 2 compares the results of heuristics miner and AGNEs on this event log.

The goal of process discovery is to give an idea of how the processes recorded in the event log *actually* have taken place. This goal makes process discovery an inherently *descriptive* learning task. To evaluate the accuracy of the discovered process model, it is therefore justified to compare the learned process models on the same sequence the process models are learned from. In the process discovery literature, this training-log-based evaluation has been the dominant evaluation paradigm (Alves de Medeiros et al., 2007; Weijters et al., 2006). Consequently, the remaining experiments of this paper use training-log-based evaluation.

6.2 Training-log-based Noise Experiment

In these experiments we have stirred up the 34 event logs with artificial noise. In the literature, six artificial noise types have been described (Maruster, 2003; Alves de Medeiros et al., 2007): (1) *missing head*: the removal of the head of a sequence, (2) *missing body*: the removal of the mid section of a sequence, (3) *missing tail*: the removal of the end section of a sequence, (4) *swap tasks*: the interchange of two random events in a sequence, (5) *remove task*: the removal of a random event in a sequence, and (6) *mix all*: a combination of all of the above. The noise has been added with the AddNoiseLogFilter event log filter available in the ProM framework. This filter has been applied after ungrouping the 34 event logs. To keep the size of the experiment under control, we

have limited the noise types used in our experiments to *mix all* and *swap tasks*. For both noise types, the used noise levels of 5%, 10%, 20% and 50% are applied.

Table 4 reports the average results of the discovered process models over the all 34 corresponding zero noise event logs. As is known from the literature, heuristics miner is resilient to noise, whereas the formal approaches of α^{+} and α^{++} and the genetic miner are known to overfit the noise in event logs. On 11 event logs from the bn1, bn2, and bn3 processes, the α^{++} implementation was incapable of producing an outcome. These missing values resulted in a score of 0 for each measure. Furthermore, the state space analysis required to calculate the behavioral appropriateness measure $a_B^{'}$ produces invalid outcomes that occur 27 times for the results of the genetic miner algorithm out of a total of 272 (34 x 8) experiments, the reported results for the genetic miner are less suitable for comparison. For this reason, we only indicate the significance of the r_B^p and s_B^n outcomes with respect to the top ranking performance. For the *PM*, f, and $a_B^{'}$ metrics, the algorithm that has obtained the best average score, is underlined. For every noise level, AGNEs obtains accuracy results that are robust and not significantly different from the results obtained by heuristics miner. This is a remarkable result, as AGNEs is a more expressive algorithm than heuristics miner, also capable of detecting more complex structures such as non-local dependencies and duplicate activities.

To calibrate the metrics, we also report their evaluation of the so-called flower model for the zero-noise case. Because the flower model parses every possible sequence, it has a perfect recall but zero specificity. These properties are to some extent reflected in the metrics in Table 4. The fitness measure f and the behavioral recall measure r_B^p are both 1.0, whereas the behavioral specificity metric s_B^n amounts to 0. The behavioral appropriateness measure $a_B^{'}$ does not really seem to quantify the lack of specificity of the flower model.

The reasons why AGNEs is robust to noise can be put down to the following. First of all, the generation of negative events is not invalidated by the presence of noise. Noise is *additional* low-frequent behavior that will result in less negative events being generated by AGNEs. However, the presence of noisy positive events does not lead to the generation of noisy negative ones. Another property that adds to robustness, is that the constraints in AGNEs ' language bias allows it to come up with structured patterns and to some extent prevents the construction of arbitrary connections between transitions, while remaining expressive with regard to short loops, duplicate tasks and non-local behavior. Finally, the formulation of process discovery as a classification allows for the application of an already robust classification algorithm (TILDE). Like many classification learners, TILDE takes into account the frequency of an anomaly, when constructing the preconditions for each activity type. Moreover, TILDE applies the same tree-level pruning method as C4.5 (Quinlan, 1993).

7. A Case Study

This section shows the result of the AGNEs process discovery algorithm applied to an event log of customer-initiated processes, recorded by a European telecom provider. The goal of the case study was to investigate whether AGNEs can be usefully applied to map the routing choices that are made between queues of a workflow management system (WfMS). With 18721 process instances and 127 queues, the obtained log file has a size of over 130 megabytes in the form of a comma-separated text file. The case study gives an idea of the scalability of the algorithm towards large event logs and the usefulness of the AGNEs process discovery algorithm on realistic, real-life processes.

		PM	*f*	a'_B	*acc*	r^p_B	s^n_B	acc^{pn}_B
zero-noise	α^+	0.72	*0.96*	**0.92**	**0.94**	*0.96*	*0.85*	*0.91*
	α^{++}	0.82	**0.98**	**0.87**	**0.92**	**0.98**	**0.93**	**0.96**
	AGNEs	0.90	**0.99**	**0.87**	**0.93**	**0.99**	**0.96**	**0.98**
	genetic	0.91	**1.00**	*0.83*	**0.91**	**0.99**	**0.95**	**0.97**
	heuristics	0.88	**0.99**	**0.86**	**0.92**	**0.99**	**0.95**	**0.97**
	flower	1.00	1.00	0.23	0.61	1.00	0.00	0.50
mix_all0.05	α^+	0.11	0.83	0.85	0.84	*0.87*	*0.62*	*0.74*
	α^{++}	0.00	0.79	0.65	0.72	*0.75*	*0.63*	*0.69*
	AGNEs	0.89	0.99	0.87	0.93	**0.99**	**0.95**	**0.97**
	genetic	0.74	0.99	0.63	0.81	**0.98**	**0.91**	**0.95**
	heuristics	0.88	0.98	0.87	0.93	**0.98**	**0.94**	**0.96**
mix_all0.1	α^+	0.08	0.80	0.84	0.82	*0.84*	*0.59*	*0.72*
	α^{++}	0.00	0.73	0.80	0.76	*0.64*	*0.64*	*0.64*
	AGNEs	0.83	0.99	0.89	0.94	**0.99**	**0.96**	**0.97**
	genetic	0.51	0.97	0.59	0.78	*0.94*	*0.78*	*0.86*
	heuristics	0.88	0.99	0.86	0.92	**0.99**	**0.95**	**0.97**
mix_all0.2	α^+	0.00	0.77	0.91	0.84	*0.82*	*0.51*	*0.67*
	α^{++}	0.00	0.65	0.65	0.65	*0.49*	*0.63*	*0.55*
	AGNEs	0.79	0.97	0.87	0.92	**0.97**	**0.94**	**0.96**
	genetic	0.47	0.96	0.53	0.74	*0.93*	*0.73*	*0.83*
	heuristics	0.86	0.98	0.85	0.92	**0.98**	**0.94**	**0.96**
mix_all0.5	α^+	0.00	0.63	0.75	0.69	*0.67*	*0.46*	*0.56*
	α^{++}	0.00	0.51	0.61	0.58	*0.26*	*0.70*	*0.48*
	AGNEs	0.54	0.96	0.77	0.87	**0.97**	**0.90**	**0.93**
	genetic	0.20	0.95	0.43	0.69	*0.86*	*0.53*	*0.69*
	heuristics	0.66	0.97	0.74	0.85	**0.96**	**0.88**	**0.92**
swap_tasks0.05	α^+	0.00	0.65	0.85	0.75	*0.76*	*0.45*	*0.60*
	α^{++}	0.00	0.59	0.67	0.63	*0.52*	*0.61*	*0.56*
	AGNEs	0.90	0.99	0.87	0.93	**0.99**	**0.96**	**0.97**
	genetic	0.44	0.95	0.61	0.78	*0.90*	*0.74*	*0.82*
	heuristics	0.88	0.99	0.85	0.92	**0.99**	**0.95**	**0.97**
swap_tasks0.1	α^+	0.00	0.58	0.86	0.72	*0.69*	*0.48*	*0.58*
	α^{++}	0.00	0.53	0.66	0.59	*0.38*	*0.61*	*0.49*
	AGNEs	0.78	0.98	0.87	0.93	**0.98**	**0.94**	**0.96**
	genetic	0.38	0.94	0.53	0.74	*0.89*	*0.65*	*0.77*
	heuristics	0.80	0.97	0.86	0.92	**0.98**	**0.94**	**0.96**
swap_tasks0.2	α^+	0.00	0.54	0.77	0.66	*0.59*	*0.52*	*0.55*
	α^{++}	0.00	0.45	0.65	0.55	*0.27*	*0.67*	*0.47*
	AGNEs	0.73	0.97	0.86	0.92	**0.98**	**0.93**	**0.95**
	genetic	0.19	0.93	0.62	0.77	*0.84*	*0.52*	*0.68*
	heuristics	0.69	0.96	0.87	0.92	**0.96**	**0.88**	**0.92**
swap_tasks0.5	α^+	0.00	0.41	0.61	0.51	*0.40*	*0.63*	*0.51*
	α^{++}	0.00	0.36	0.61	0.48	*0.16*	**0.77**	*0.46*
	AGNEs	0.32	0.91	0.72	0.81	**0.95**	**0.82**	**0.89**
	genetic	0.07	0.93	0.77	0.85	*0.79*	*0.40*	*0.59*
	heuristics	0.45	0.94	0.66	0.80	**0.94**	**0.83**	**0.89**

Table 4: noise experiments - average, zero-noise training-log-based results

The event log consists of events about customer-initiated processes that are handled at three different locations by the employees of the telecom provider. The handling of cases is organized in a first line and a second line. First-line operators are junior operators that deal with frequent customer requests for which standardized procedures have been put in place. When a first-line operator cannot process a case, it is routed to a queue of the second line. Second-line case handling is operated by senior experts who have the authority to make decisions to solve the more involved cases. The

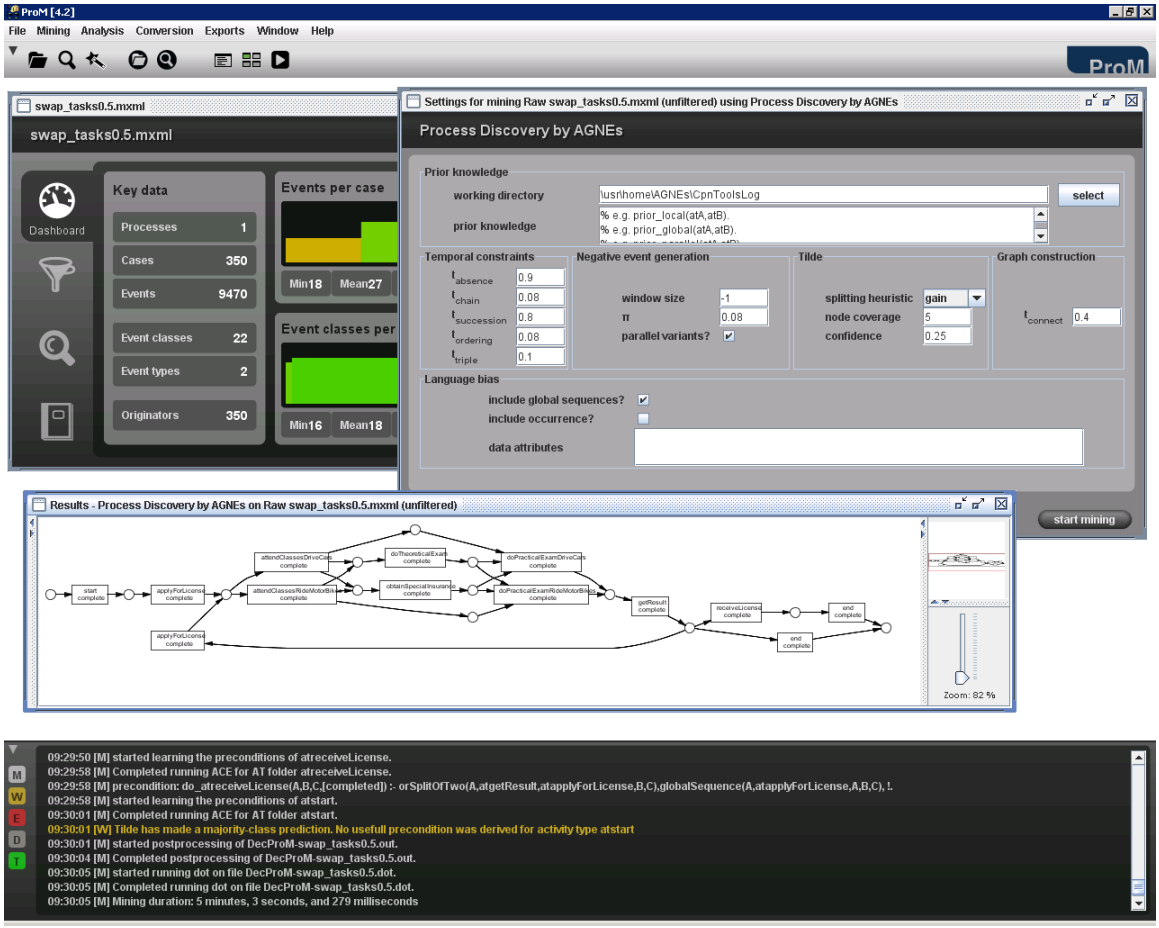


Figure 3: AGNEs in ProM 4.2

second-line processes are coordinated and supported by means of a workflow management system (WfMS). The obtained event log consists of these second-line case handling events. The second-line WfMS is organized as a system of 127 logical queues. Each queue corresponds to a number of similar types of activities that are to be carried out. At any given moment each active case resides in exactly one queue. Employees can process a case by taking it out of the queue into their personal work bin. Every evolution of a case is documented by adding notes. Moreover, employees can classify the nature of the case according to a number of data fields. In addition, a worker or dispatcher has the ability to reroute cases to different queues whenever this is necessary. The system imposes no restrictions with regard to the routing of cases. Queues represent a work distribution system and are akin to roles in WfMS. For the purpose of this analysis, queues are considered to be activity types. The 40 most frequently occurring queues were retained for further analysis. Nine process instances that did not involve at least one of these 40 queues were retained from the event log.

We compare the mining results of AGNEs, genetic miner, and heuristics miner. Some parameter settings that are different from the experimental evaluation in the previous section. In particular, AGNEs was provided with the prior knowledge that no activity can occur concurrently: $\forall a, b \in A$:

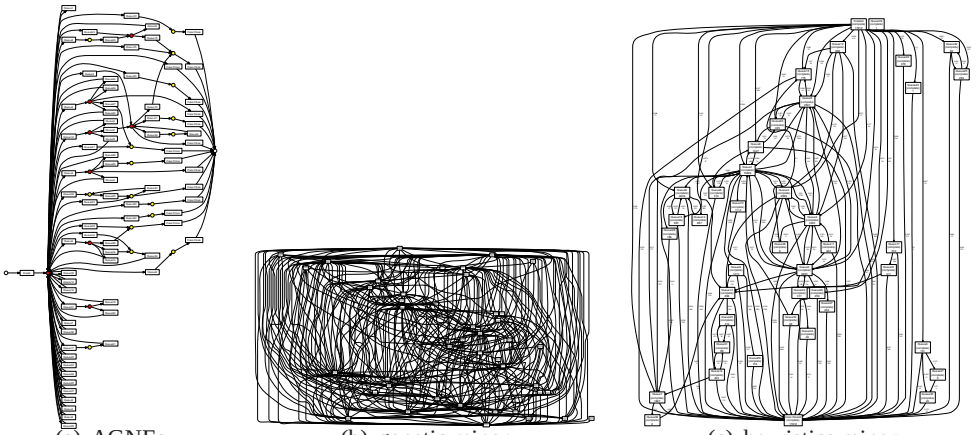
	PM	f	$a_B^{'}$	acc	r_B^p	s_B^n	acc_B^{pn}
AGNEs	0.06	0.94	0.67	0.80	0.93	0.67	0.80
genetic	0.03	x.xx	x.xx	x.xx	0.83	0.62	0.73
heuristics	0.80	0.97	0.72	0.85	0.96	0.88	0.92
flower model	0.00	1.00	0.76	0.88	1.00	0.00	0.50

Table 5: telecom—training-log-based results

$PriorSerial(a,b)$. This prior knowledge is justifiable, as no case can be routed to or reside in several queues at the same time. Genetic miner has been running for 5000 generations, with a population size of 10. These parameter settings correspond to the parameter settings in the case study described by Alves de Medeiros et al. (2007). To account for the prior knowledge that no concurrent behavior is contained in the event log, heuristics miner needs to have an infinite AND threshold. In general AGNEs allows to provide a-priori locality and parallelism information for individual pairs of activity types. This fine-grained a-priori knowledge cannot be provided by fine-tuning the AND threshold with heuristics miner. Currently, it is not possible to constrain the search space of genetic miner with this a-priori knowledge.

The results of applying these process discovery algorithms on the filtered event log are displayed in Figure 4. Table 5 gives an overview of the metrics that compare the discovered process models to the original event log. As the purpose of the case study is to provide the most accurate description of the event log, the use of training data for evaluation is justified. To calibrate the metrics, we also report their evaluation of the so-called flower model. Because the flower model represents random behavior, it has a perfect recall of the all behavior in the event log but it also has much *additional* behavior compared to the event log. Because of the latter fact, the flower model has zero specificity. These properties are to some extent reflected in the metrics in Table 5. The fitness measure f and the behavioral recall measure r_B^p are both 1.0, whereas the behavioral specificity metric s_B^n amounts to 0. The table also indicates the usefulness of the metrics proposed in this paper. The parsing measure PM does not reflect the recall of the flower model. Furthermore, the behavioral appropriateness measure $a_B^{'}$ does not really seem to quantify the lack of specificity of the flower model. For the genetic miner mining result, the ProM implementations of f, and $a_B^{'}$ did not produce an outcome. To calculate these metrics, a conversion of the heuristics nets into Petri nets is required. The resulting Petri nets, which have many invisible transitions, are seemingly too complex to calculate the metric. These results are an indication of the usefulness of the new specificity metric proposed in this paper.

Comparing the available r_B^p and s_B^n outcomes, it can be observed that AGNEs performs better than genetic miner, but worse than heuristics miner on the obtained event log. The discovered process model by AGNEs has an accuracy of 80%, whereas the models discovered by genetic miner and heuristics miner have an accuracy of 73% and 92% respectively. This case study brings forward that human-centric processes contained in the event log can take place in a less structured fashion than often is assumed by process discovery algorithms. In this particular case, for instance, it seems that OR-splits and OR-joins can involve a rather high number of outgoing or incoming activities. In the current implementation, the language bias of TILDE is limited to conjunctions and disjunctions of *NS* constructs of length two and three. This imposes limitations on the hypotheses that can be learned by AGNEs. As indicated in Section 4.3, solving this language bias issue, requires construct-



(a) AGNEs (b) genetic miner (c) heuristics miner

Figure 4: telecom—mining results

ing a proprietary ILP classification algorithm that during each refinement step allows considering conjunctions of *NS* constructs of variable lengths. In this regard, the language bias of Heuristics Miner seems to be less limiting, although Heuristics Miner does not have many of the declarative properties of AGNEs. Another outcome of the case study, is that the proposed measures for behavioral recall r_B^p and behavioral specificity s_B^n in practice turn out to be valuable metrics for assessing the accuracy of a discovered process model.

8. Related Work

Process discovery can be seen as an application of the machine learning of grammars from positive data, of which Angluin and Smith (1983) provide an overview. Gold (1967) has shown that important classes of recursively enumerable languages cannot be *identified in the limit* from a finite number of positive examples only. Instead, both positive and negative examples are required for grammar learning to distinguish the right hypothesis among an infinite number of grammars that fit the positive examples. Whereas Gold's negative learnability result applies to the learning of grammars with perfect accuracy, process discovery is more concerned with the ability to discover process models that have *only* a good recall and specificity. Learning grammars from only positive examples requires a tradeoff between overly general and overly specific hypotheses. Muggleton (1996) shows that in a Bayesian framework, logic programs are learnable with arbitrarily low error from positive examples only. Bayes' theorem allows to formulate this tradeoff as a tradeoff between size and generality of the hypotheses and learning can be considered to maximization the posterior probability over all hypotheses. In this paper, a new approach for making the tradeoff between generality and specificity is proposed, by inducing artificial negative events using a (highly configurable) assumption about the completeness of the behavior displayed by the positive examples in the event log. Another difference with grammar learning is that in grammar learning, the hypothesis space is often expressed as production rules, automata or regular expressions, whereas process discovery uses formalisms that can represent the concurrency and synchronization concerns of processes more elegantly.

Process models are in general deterministic. In the literature, there are many formalisms to represent and learn the probability distributions of *stochastic*, generative grammars over sequences of observed characters and unobserved state variables. Historically, techniques like Markov models, hidden Markov models, factorial hidden Markov models, and dynamic Bayesian networks have been first applied to speech recognition, and bio-informatics (Durbin et al., 1998). Each representation has its own particular modeling features that makes it more or less suited for representing the human-centric behavior of business processes. Factorial hidden Markov models, for instance, have a distributed state representation that allows for the modeling of concurrent behavior (Ghahramani and Jordan, 1997). Hidden Markov models have furthermore been provided with a first-order, extension that allows for the representation of sequences of logical atoms rather than alphabets of flat characters (Kersting et al., 2006). Other authors describe learning mixture models to identify meaningful clusters of (hidden) Markov models (Smyth, 1997; Cadez et al., 2003). Whereas stochastic models provide useful information, their probabilistic nature tends to compromise the comprehensibility of discovered process models. Business processes have well-defined start, end, split, and synchronization nodes. The network structure of stochastic models does not visualize this. For instance, although hidden states could be useful in representing duplicate activities—the same activity label is logged in different contexts—a hidden Markov model is unlikely to be capable of comprehensively representing its different usage contexts.

In contrast, Mannila and Meek (2000) describe a technique to learn two-component mixture models of global partial orders that provide an understandable, global view of the sequential data. The authors assume the presence of one dominant, global partial order and consider a generic partial order with random behavior to deal with low-frequent variations (noise) from the former model. Silva et al. (2005) describe a probabilistic model and algorithm for process discovery that discovers so-called and/or graphs in polynomial time. These and/or graphs are comprehensible, directed acyclic graphs that have the advantage over global partial order representations that they can differentiate between parallel and serial split and join points. Pei et al. (2006) describe a scalable technique for discovering the complete set of frequent, closed partial orders from sequential data. The three aforementioned techniques assume each item to occur only once within a sequence, and do not consider recurrent behavior (cycles), nor duplicate activities.

The term process discovery was coined by Cook and Wolf (1998), who apply it in the field of software engineering. Their Markov algorithm can only discover sequential patterns as Markov chains cannot elegantly represent concurrent behavior. The idea of applying process discovery in the context of workflow management systems stems from Agrawal et al. (1998) and Lyytinen et al. (1998). The value of process discovery for the general purpose of process mining (van der Aalst et al., 2007) is well illustrated by the plugins within the ProM framework. In analogy with the WEKA toolset for data mining (Witten and Frank, 2000), the ProM Framework consists of a large number of plugins for the analysis of event logs (Process Mining Group, TU/Eindhoven, 2008). The *Conformance Checker* plugin (Rozinat and van der Aalst, 2008), for instance, allows identifying the discrepancies between an idealized process model and an event log. Moreover, with a model that accurately describes the event log, it becomes possible to use the time-information in an event log for the purpose of performance analysis, using, for instance, the *Performance Analysis with Petri nets* plugin.

Table 6 provides a chronological overview of process discovery algorithms that have been applied to the context of workflow management systems. The α algorithm can be considered to be a theoretical learner for which van der Aalst et al. (2004) prove that it can learn an important class

of workflow nets, called structured workflow nets, from complete event logs. The α algorithm assumes event logs to be complete with respect to all allowable binary sequences and assumes that the event log does not contain any noise. Therefore, the α algorithm is sensitive to noise and incompleteness of event logs. Moreover, the original α algorithm was incapable of discovering short loops or non-local, non-free choice constructs. Alves de Medeiros et al. (2004) have extended the α algorithm to mine short loops and called it α^+. Wen et al. (2007) made an extension for detecting implicit dependencies, for detecting non-local, non-free choice constructs. None of the algorithms can detect duplicate activities. The main reason why the α algorithms are sensitive to noise, is that they does not take into account the frequency of binary sequences that occur in event logs. Weijters and van der Aalst (2003) and Weijters et al. (2006) have developed a robust, heuristic-based method for process discovery, called heuristics miner, that is known to be noise resilient. Heuristics miner can discover short loops, and non-local, non-free choice as it can consider non-local dependencies within an event log. However, heuristics miner cannot detect duplicate activities.

Algorithm (Ref.)	Summary
global partial orders (Mannila and Meek, 2000)	Learns a two-component mixture model of a dominant series-parallel partial order and a trivial partial order by searching for the dominant partial order that yields the highest probability for generating the observed sequence database.
little thumb, **heuristics miner** (Weijters and van der Aalst, 2003) (Weijters et al., 2006)	Extends the α algorithm by taking into account the frequency of the follows relationship, to calculate dependency/frequency tables from the event log and uses heuristics to convert this information into a heuristics net.
α, α^+ (van der Aalst et al., 2004) (Alves de Medeiros et al., 2004)	Derives a Petri net from local, binary ordering relations detected within an event log.
splitpar—InWoLvE (Herbst and Karagiannis, 2004)	Derives a so-called stochastic activity graph and converts it into a structured process model in the Adonis Modeling language.
multi-phase miner (van Dongen and van der Aalst, 2005b)	Constructs a process model for every sequence in the log and aggregates the model into an event-driven process chain.
α^{++} (Wen et al., 2007)	Extends the α algorithm to discover non-local, non-free choice constructs.
– (Silva et al., 2005)	A probabilistic approach to process discovery.
frecpo (Pei et al., 2006)	A scalable technique for discovering the complete set of frequent, closed partial orders from sequential data.
FSM/Petrify miner (van der Aalst et al., 2006)	Derives a highly configurable finite state machine from the event log and folds the finite state machine into regions using the theory of regions.
– (Ferreira and Ferreira, 2006)	Learns the case data preconditions and effects of activities with ILP classification techniques and user-supplied negative events.
genetic miner (Alves de Medeiros et al., 2007)	A genetic algorithm that selects the more complete and precise heuristics nets over generations of nets.
DecMiner (Lamma et al., 2007)	A classification technique that learns the preconditions of activities with the ICL ILP learner from event logs with user-supplied negative sequences.
fuzzy miner (Günther and van der Aalst, 2007)	An adaptive simplification and visualization technique based on significance and correlation measures to visualize unstructured processes.

Table 6: Chronological overview of process discovery algorithms

van Dongen and van der Aalst (2005b) present a multi-phase approach to process mining that starts from the individual process sequences, constructs so-called instance graphs for each sequence that account for parallelism, and then aggregates these instance graphs according to previously detected binary relationships between activity types. Interestingly, the aggregation ensures that every discovered process model has a perfect recall, but generally scores less on specificity. Herbst and Karagiannis (2004) describe the working of the splitpar algorithm that is part of the InWoLvE framework for process analysis. This algorithm derives a so-called stochastic activity graph and

converts it into a structured process model. The splitpar algorithm is capable of detecting duplicate activities, but it is not able to discover non-local dependencies.

Alves de Medeiros et al. (2007) describe a genetic algorithm for process discovery. The fitness function of this genetic algorithm incorporates both a recall and a specificity measure that drives the genetic algorithm towards suitable models. The genetic miner is capable of detecting non-local patterns in the event log and is described to be fairly robust to noise. In her PhD, Alves de Medeiros (2006) describes an extension to this algorithm for the discovery of duplicate tasks.

van der Aalst et al. (2006) present a two-phase approach to process discovery that allows to configure when states or state transitions are considered to be similar. The ability to manipulate similarity is a good approach to deal with incomplete event logs. In particular, several criteria can be considered for defining similarity of behavior and states: the inclusion of future or past events, the maximum horizon, the activities that determine state, whether ordering matters, the activities that visibly can bring about state changes, etcetera. Using these criteria, a configurable finite state machine can be constructed from the event log. In a second phase, the finite state machine is folded into regions using the existing theory of regions (Cortadella et al., 1998). For the moment, the second phase of the algorithm still poses difficulties with respect to constructing suitable process models. The approach presented in this paper considers window size (maximum horizon) and parallel variants as similarity criteria when generating artificial negative events.

Günther and van der Aalst (2007) present an adaptive simplification and visualization technique based on significance and correlation measures to visualize the behavior in event logs at various levels of abstraction. The contribution of this approach is that it can also be applied to less structured, or unstructured processes of which the event logs cannot easily be summarized in concise, structured process models.

Several authors have used classification techniques for the purpose of process discovery. Maruster et al. (2006), for instance, were among the first to investigate the use of rule-induction to predict dependency relationships between activities from a corpus of reference logs that portray various levels of noise and imbalance. To this end, the authors use a propositional rule induction technique, the uni-relational classification learner RIPPER (Cohen, 1995), on a table of direct metrics for each process task in relation to each other process task, which is generated in a pre-processing step.

Ferreira and Ferreira (2006) apply a combination of ILP learning and partial-order planning techniques to process mining. Rather than generating artificial negative events, negative examples are collected from the users who indicate whether a proposed execution plan is feasible or not. By iteratively combining planning and learning, a process model is discovered that is represented in terms of the case data preconditions and effects of its activities. In addition to this new process mining technique, the contribution of this work is in the truly integrated BPM life cycle of process generation, execution, re-planning and learning. Lamma et al. (2007) also describe the use of ILP to process mining. The authors assume the presence of negative sequences to guide the search algorithm. Unlike the approach of Ferreira and Ferreira, who use partial-order planning to present the user with an execution plan to accept or reject (a negative example), this approach does not provide an immediate answer to the origin of the negative events. Contrary to our approach, the latter two approaches are not concerned with the construction of a graphical, control-flow based process model and do not consider the generation of artificial negative events.

9. Conclusion

Process discovery aims at accurately summarizing an event log in a structured process model. So far, the discovery of structured processes by supplementing event logs with *artificial* negative events has not been considered in the literature. The advantage is that it allows representing process discovery as a multi-relational classification problem to which existing classification learners can be applied. In this paper, the generation of artificial negative events gives rise to a new process discovery algorithm and to new metrics for quantifying the recall and specificity of a process model vis-à-vis an event log.

Process discovery algorithms must deal with challenges such as expressiveness, noise, incomplete event logs and the inclusion of prior knowledge. Dealing with one challenge sometimes leads to poor performance with respect to another. The technique presented in this paper, simultaneously addresses many of these challenges. This can be concluded from the results of a large benchmark study applied to AGNEs and four state-of-the art process discovery algorithms. A comparative benchmark study of this scale is the first in the field of process discovery. The benchmark experiments indicate that our technique can discover complex structures such as short loops, duplicate activities, and non-free choice constructs, while remaining robust to noise. In addition, our technique has a new, declarative way of dealing with incomplete event logs that diminishes the effects of concurrent and recurrent behavior on the generation of artificial negative events. Finally, our technique is capable of having prior knowledge constrain the hypothesis space during process discovery. These declarative aspects—the inclusion of prior knowledge, the configurability of the negative event generation procedure and the language bias—potentially make it very useful in practical applications. Another outcome of the benchmark study is the usefulness of the new specificity metric, which in contrast to existing metrics can always be calculated and produces intuitive results.

Acknowledgments

We extend our gratitude to the guest editors, the anonymous reviewers, and the production editor, as their many constructive and detailed remarks certainly contributed much to the quality of this paper. Further, we would like to thank the Flemish Research Council for financial support (FWO postdoctoral research grant, Odysseus grant B.0915.09).

References

Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. Mining process models from workflow logs. In *Proceedings of the 6th International Conference on Extending Database Technology (EDBT'98)*, volume 1377 of *Lecture Notes in Computer Science*, pages 469–483. Springer, 1998.

Ana Karla Alves de Medeiros. *Genetic Process Mining*. PhD thesis, Technische Universiteit Eindhoven, 2006.

Ana Karla Alves de Medeiros, Boudewijn F. van Dongen, Wil M. P. van der Aalst, and Anton J. M. M. Weijters. Process mining: Extending the alpha-algorithm to mine short loops. BETA working paper series 113, Eindhoven University of Technology, 2004.

Ana Karla Alves de Medeiros, Anton J. M. M. Weijters, and Wil M. P. van der Aalst. Genetic process mining: an experimental evaluation. *Data Mining and Knowledge Discovery*, 14(2): 245–304, 2007.

Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys*, 15(3):237–269, 1983.

Hendrik Blockeel. *Top-Down Induction of First-Order Logical Decision Trees*. PhD thesis, Katholieke Universiteit Leuven, 1998.

Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.

Hendrik Blockeel, Luc Dehaspe, Bart Demoen, Gerda Janssens, Jan Ramon, and Henk Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.

Igor Cadez, David Heckerman, Christopher Meek, Padhraic Smyth, and Steven White. Model-based clustering and visualization of navigation patterns on a web site. *Data Mining and Knowledge Discovery*, 7(4):399–424, 2003. ISSN 1384-5810.

William W. Cohen. Fast effective rule induction. In *Proceedings of the 12th International Conference on Machine Learning*, pages 115–123, Tahoe City, CA, 1995. Morgan Kaufmann Publishers.

Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alexandre Yakovlev. Deriving petri nets from finite transition systems. *IEEE Transactions on Computers*, 47(8):859–882, 1998.

Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge university press, 1998.

Sašo Džeroski. Multi-relational data mining: an introduction. *SIGKDD Explorations*, 5(1):1–16, 2003.

Sašo Džeroski and Nada Lavrač. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York, 1994.

Sašo Džeroski and Nada Lavrač, editors. *Relational Data Mining*. Springer-Verlag, Berlin, 2001.

Richard Dybowski, Kathryn B. Laskey, James W. Myers, and Simon Parsons. Introduction to the special issue on the fusion of domain knowledge with data for decision support. *Journal of Machine Learning Research*, 4:293–294, 2003.

Hugo Ferreira and Diogo R. Ferreira. An integrated life cycle for workflow management based on learning and planning. *International Journal of Cooperative Information Systems*, 15(4):485–505, 2006.

Zoubin Ghahramani and Michael I. Jordan. Factorial hidden markov models. *Machine Learning*, 29(2-3):245–273, 1997.

Stijn Goedertier. *Declarative Techniques for Modeling and Mining Business Processes*. Phd thesis, Katholieke Universiteit Leuven, Faculty of Business and Economics, Leuven, September 2008.

Stijn Goedertier, David Martens, Bart Baesens, Raf Haesen, and Jan Vanthienen. Process mining as first-order classification learning on logs with negative events. In *Proceedings of the 3rd Workshop on Business Processes Intelligence (BPI'07)*, volume 4928 of *Lecture Notes in Computer Science*. Springer, 2008.

E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

Christian W. Günther and Wil M. P. van der Aalst. Fuzzy mining - adaptive process simplification based on multi-perspective metrics. In *Proceedings of the 5th International Conference on Business Process Management, BPM 2007*, volume 4714 of *Lecture Notes in Computer Science*, pages 328–343. Springer, 2007.

Joachim Herbst and Dimitris Karagiannis. Workflow mining with InWoLvE. *Computers in Industry*, 53(3):245–264, 2004.

Kristian Kersting, Luc De Raedt, and Tapani Raiko. Logical hidden markov models. *Journal of Artificial Intelligence Research*, 25:425–456, 2006.

Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari. Inducing declarative logic-based models from labeled traces. In *Proceedings of the 5th International Conference on Business Process Management, BPM 2007*, volume 4714 of *Lecture Notes in Computer Science*, pages 344–359. Springer, 2007.

Kalle Lyytinen, Lars Mathiassen, Janne Ropponen, and Anindya Datta. Automating the discovery of as-is business process models: Probabilistic and algorithmic approaches. *Information Systems Research*, 9(3):275–301, 1998.

Heikki Mannila and Christopher Meek. Global partial orders from sequential data. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '00)*, pages 161–168, New York, NY, USA, 2000. ACM.

Laura Maruster. *A Machine Learning Approach to Understand Business Processes*. PhD thesis, Eindhoven University of Technology, Eindhoven, 2003.

Laura Maruster, Anton J. M. M. Weijters, Wil M. P. van der Aalst, and Antal van den Bosch. 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.

Stephen Muggleton. Inductive logic programming. In *Proceedings of the 1st International Conference on Algorithmic Learning Theory*, pages 42–62, 1990.

Stephen Muggleton. Learning from positive data. In Stephen Muggleton, editor, *Inductive Logic Programming Workshop*, volume 1314 of *Lecture Notes in Artificial Intelligence*, pages 358–376. Springer, 1996.

Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4): 541–580, 1989.

Jian Pei, Haixun Wang, Jian Liu, Ke Wang, Jianyong Wang, and Philip S. Yu. Discovering frequent closed partial orders from strings. *IEEE Transactions on Knowledge and Data Engineering*, 18 (11):1467–1481, 2006.

Process Mining Group, TU/Eindhoven. Process mining web page: research, tools and application. `http://processmining.org`, 2008.

J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

Anne Rozinat and Wil M. P. van der Aalst. Conformance checking of processes based on monitoring real behavior. *Information Systems*, 33(1):64–95, 2008.

Anne Rozinat, Ana Karla Alves De Medeiros, Christian W. Günther, Anton J. M. M. Weijters, and Wil M. P. van der Aalst. Towards an evaluation framework for process mining algorithms. BETA working paper series 224, Eindhoven University of Technology, 2007.

Ricardo Silva, Jiji Zhang, and James G. Shanahan. Probabilistic workflow mining. In *KDD '05: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 275–284, New York, NY, USA, 2005. ACM.

Padhraic Smyth. Clustering sequences with hidden markov models. In *Advances in Neural Information Processing Systems*, volume 9, pages 648–654. MIT Press, 1997.

Wil M. P. van der Aalst. Verification of workflow nets. In *Proceedings of the 18th International Conference on the Application and Theory of Petri Nets 1997 (ICATPN '97)*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer, 1997.

Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

Wil M. P. van der Aalst, Anton J. M. M. Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

Wil M. P. van der Aalst, Vladimir Rubin, Boudewijn F. van Dongen, Ekkart Kindler, , and Christian W. Günther. Process mining: A two-step approach using transition systems and regions. BPM-06-30, BPM Center Report, 2006.

Wil M. P. van der Aalst, Hajo A. Reijers, Anton J. M. M. Weijters, Boudewijn F. van Dongen, Ana Karla Alves de Medeiros, Minseok Song, and H. M. W. (Eric) Verbeek. Business process mining: An industrial application. *Information Systems*, 32(5):713–732, 2007.

Boudewijn F. van Dongen and Wil M. P. van der Aalst. A meta model for process mining data. In *EMOI-INTEROP*, volume 160 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005a.

Boudewijn F. van Dongen and Wil M. P. van der Aalst. Multi-phase process mining: Aggregating instance graphs into EPCs and Petri nets. In *Proceedings of the 2nd International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management (PNCWB)*, 2005b.

Tony Van Gestel, Johan A.K. Suykens, Bart Baesens, Stijn Viaene, Jan Vanthienen, Guido Dedene, B. De Moor, and J. Vandewalle. Benchmarking least squares support vector machine classifiers. *Machine Learning*, 54(1):5–32, 2004.

Anton J. M. M. Weijters and Wil M. P. van der Aalst. Rediscovering workflow models from event-based data using little thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.

Anton J. M. M. Weijters, Wil M. P. van der Aalst, and Ana Karla Alves de Medeiros. Process mining with the heuristics miner-algorithm. BETA working paper series 166, Eindhoven University of Technology, 2006.

Lijie Wen, Wil M. P. van der Aalst, Jianmin Wang, and Jiaguang Sun. Mining process models with non-free-choice constructs. *Data Mining and Knowledge Discovery*, 15(2):145–180, 2007.

Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.