Genetic Process Mining: An Experimental Evaluation

A.K. Alves de Medeiros [*], A.J.M.M. Weijters
and W.M.P. van der Aalst

*Department of Technology Management, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.*

Abstract

One of the aims of process mining is to retrieve a process model from an event log. The discovered models can be used as *objective* starting points during the deployment of process-aware information systems (PAIS) [19] and/or as a feedback mechanism to check prescribed models against enacted ones. However, current techniques have problems when mining processes that contain non-trivial constructs and/or when dealing with the presence of noise in the logs. Most of the problems happen because many current techniques are based on *local* information in the event log. To overcome these problems, we try to use genetic algorithms to mine process models. The main motivation is to benefit from the *global search* performed by this kind of algorithms. The non-trivial constructs are tackled by choosing an internal representation that supports them. The problem of noise is naturally tackled by the genetic algorithm because, per definition, these algorithms are robust to noise. The main challenge in a genetic approach is the definition of a good fitness measure because it guides the global search performed by the genetic algorithm. This paper explains how the genetic algorithm works. Experiments with synthetic and real-life logs show that the fitness measure indeed leads to the mining of process models that are *complete* (can reproduce all the behavior in the log) and *precise* (do not allow for extra behavior that cannot be derived from the event log). The genetic algorithm is implemented as a plug-in in the ProM framework.

Key words: process mining, genetic mining, genetic algorithms, Petri nets, workflow nets.

[*] Corresponding author.
Email address: `a.k.medeiros@tm.tue.nl` (A.K. Alves de Medeiros).

Preprint accepted by Journal of Data Mining and Knowledge Discovery *on 20/11/2006.*

1 Introduction

Today's organizations are supported by a wide variety of information systems. Some systems only support a single task (e.g., a text editor). However, most organizations are using systems that support processes, i.e., not a single task but the glue between tasks. Examples are WorkFlow Management (WFM) systems and Enterprise Resource Planning (ERP) systems. Typically, these systems record events that can be linked to the execution of some task in the process. Therefore, it makes sense to analyze these events to get feedback about enacted processes. Buzzwords such as Business Process Intelligence (BPI) and Business Activity Monitoring (BAM) indicate the interest of organizations and software developers in solutions able to extract knowledge from so-called event logs. However, most of the commercial systems (e.g., Cognos and Business Objects) focus on exclusively performance issues such as flow time and utilization. These systems abstract from the process itself and can only be applied if the process is well-defined and fixed. ARIS PPM is one of the few commercial systems actually trying to discover more information by monitoring events. One of the reasons for this limited support is that it is very difficult to extract process knowledge without having some a-priori process model. This triggered the development of process mining techniques that aim at automatically discovering process models based on event logs.

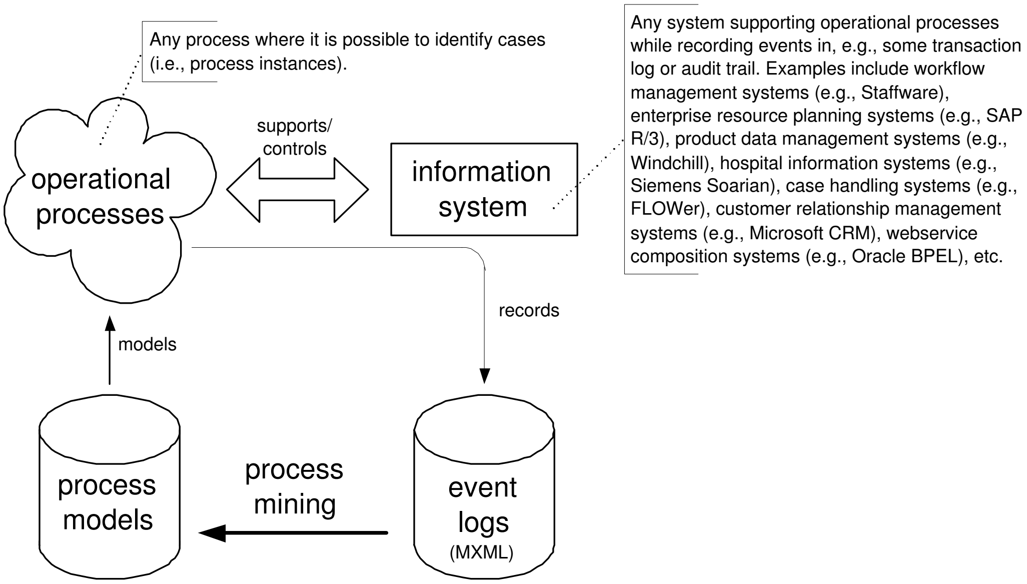

Any process where it is possible to identify cases (i.e., process instances).
Any system supporting operational processes while recording events in, e.g., some transaction log or audit trail. Examples include workflow management systems (e.g., Staffware), enterprise resource planning systems (e.g., SAP R/3), product data management systems (e.g., Windchill), hospital information systems (e.g., Siemens Soarian), case handling systems (e.g., FLOWer), customer relationship management systems (e.g., Microsoft CRM), webservice composition systems (e.g., Oracle BPEL), etc.
operational processes
supports/ controls
information system
records
models
process models
process mining
event logs
(MXML)

Fig. 1. Overview of process mining.

Figure 1 illustrates the concept of process mining. Some operational process is supported by some information system that records events in some event log. This event log is used to extract process models that describe the observed behavior. This information is valuable to better understand processes and to improve them. In our experience, real processes tend to deviate from the idealistic processes people have in mind. The practical relevance of process mining is obvious. Unfortunately, existing techniques have severe limitations [1] . Therefore, we present a new approach using a genetic algorithm. However, before we introduce our approach, we first need to clarify the concept of process mining.

1.1 Process Mining

One of the aims of process mining is to *automatically* build a process model that describes the behavior contained in an event log. The models mined by process mining tools can be used as an *objective starting point* during the deployment of systems that support the execution of processes and/or as a *feedback mechanism* to check the prescribed process model against the enacted one. We use an example to illustrate how process mining techniques work. Consider the event log shown in Table 1. This log shows the event traces (process instances) for four different applications to get a license to ride motorbikes or drive cars. Note that applicants for different types of licenses do the same theoretical exam (task "Do Theoretical Exam") but different practical ones (tasks "Do Practical Exam Drive Cars" or "Do Practical Exam Ride Motobikes"). In other words, whenever the task "Attend classes Drive Cars" is executed, the task "Do practical Exam Drive Cars" is the only one that can be executed after the applicant has done the theoretical exam. This shows that there is a *non-local* dependency between the tasks "Attend Classes Drive Cars" and "Do Practical Exam Drive Cars", and also between the tasks "Attend Classes Ride Motorbikes" and "Do Practical Exam Ride Motorbikes". The dependency is non-local because it cannot be detected by simply looking at the direct predecessor and successor of those tasks in the log in Table 1, e.g. "Attend Classes Drive Cars" is never followed directly by "Do Practical Exam Drive Cars". Moreover, note that only in some process instances (2 and 3) the task "Receive License" was executed. These process instances point to the cases in which the candidate passed the exams. Based on this log and these observations, process mining tools could be used to retrieve the model in Figure 2. In this case, we are using Petri nets [18,47] to depict this model. We do so because Petri nets will be used to explain the semantics of our internal representation. Moreover, we use Petri-net-based analysis techniques to analyse the resulting models. Using the Petri net representation, our tools allow for the automatic translation of the discovered model to a variety of modelling notations including Event-Driven Process Chains (used by ARIS, ARIS PPM, SAP) and YAWL (an open source workflow system).

[1] On Section 1.2 we elaborate more on these limitations.

Identifier	Process instance
1	Start, Apply for License, Attend Classes Drive Cars, Do Theoretical Exam, Do Practical Exam Drive Cars, Get Result, End
2	Start, Apply for License, Attend Classes Ride Motorbikes, Do Theoretical Exam, Do Practical Exam Ride Motorbikes, Get Result, Receive License, End
3	Start, Apply for License, Attend Classes Drive Cars, Do Theoretical Exam, Do Practical Exam Drive Cars, Get Result, Receive License, End
4	Start, Apply for License, Attend Classes Ride Motorbikes, Do Theoretical Exam, Do Practical Exam Ride Motorbikes, Get Result, End

Table 1
Example of an event log with 4 process instances.

Petri nets are a formalism to model concurrent processes. Graphically, Petri nets are bipartite directed graphs with two node types: *places* and *transitions*. Places represent conditions in the process. Transitions represent actions. Tasks in the event logs correspond to transitions in Petri nets. The state of a Petri net (or process for us) is described by adding tokens (black dots) to places. The dynamics of the Petri net is determined by the *firing rule*. A transition can be executed (i.e. an action can take place in the process) when all of its input places (i.e. pre-conditions) have at least a number of tokens that is equal to the number of directed arcs from the place to the transition. After execution, the transition removes tokens from the input places (one token is removed for every input arc from the place to the transition) and produces tokens for the output places (again, one token is produced for every output arc). Besides, the Petri nets that we consider have a single *start* place and a single *end* place. This means that the processes we describe have a single start point and a single end point. For the Petri net in Figure 2, in the initial state there is only one token in place “p1”. This implies that “Start” is the only transition that can be executed in the initial state. When “Start” executes (or fires), one token is removed from the place “p1” and one token is added to the place “p2”. In a similar way, the firing of “Apply for License” marks place “p3”. In this marking, “Attend Classes Drive Cars” or “Attend Classes Ride Motorbikes” can fire. If “Attend Classes Drive Cars” fires, it consumes the token in “p3” and produces one token for “p4” and another for “p5”. Note that, although the place “p5” has now one token, the transition “Do Practical

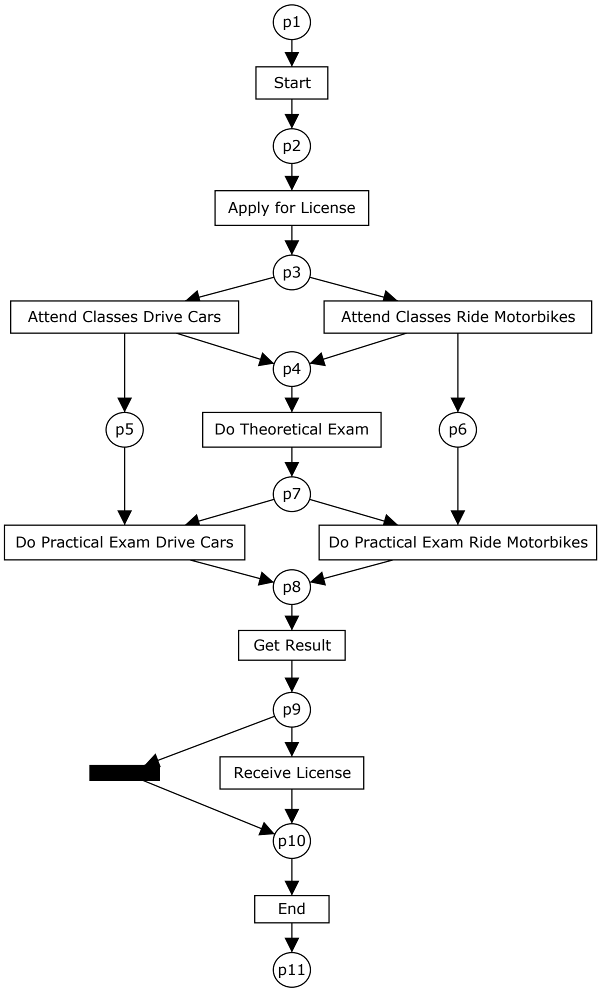

p1
Start
p2
Apply for License
p3
Attend Classes Drive Cars
Attend Classes Ride Motorbikes
p4
p5
Do Theoretical Exam
p6
p7
Do Practical Exam Drive Cars
Do Practical Exam Ride Motorbikes
p8
Get Result
p9
Receive License
p10
End
p11

Fig. 2. Mined net for the log in Table 1.

Exam Drive Cars" cannot fire yet because the place "p7" is not marked. The enabling and firing of transitions proceeds in a similar way until the place "p11" is marked.

1.2 Limitations of Current Approaches

Current research in process mining [3,5,6,24,32,10,14,15,56,60] still has problems to discover process models with certain structural constructs and/or to deal with the presence of noise in the logs (cf. Section 9). The main problematic constructs are: non-free-choice, invisible tasks and duplicate tasks [14]. *Non-free-choice* constructs combine synchronization and choice. The example in Figure 2 illustrates a non-free-choice construct involving the tasks "Do Practical Exam Drive Cars" and "Do Practical Exam Ride Motorbikes". The current techniques do not capture the dependency between (i) the tasks "Attend Classes Drive Cars" and "Do Practical Exam Drive Cars", and (ii) the

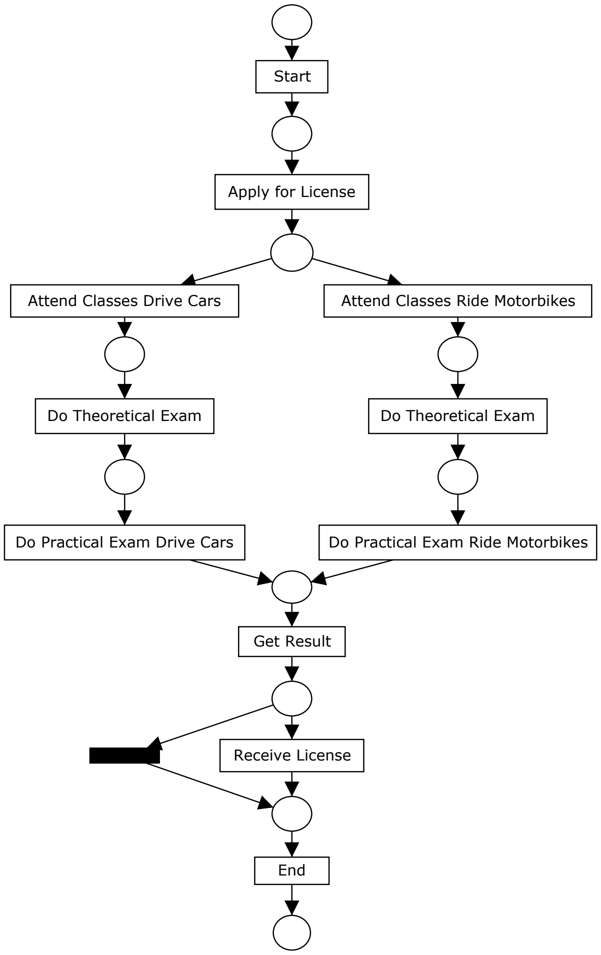

Start
Apply for License
Attend Classes Drive Cars
Attend Classes Ride Motorbikes
Do Theoretical Exam
Do Theoretical Exam
Do Practical Exam Drive Cars
Do Practical Exam Ride Motorbikes
Get Result
Receive License
End

Fig. 3. Another model that correctly portraits the behavior in the log in Table 1. Note that this model uses *duplicate tasks* instead of the *non-free-choice construct* in Figure 2.

tasks “Attend Classes Ride Motorbikes” and “Do Practical Exam Ride Motorbikes”. *Invisible tasks* are only used for routing purposes and do not appear in the log. For instance, the process in Figure 2 has an invisible task to skip the execution of the task “Receive License”. Current techniques have difficulties discovering these routing tasks because they do not appear in the log. *Duplicate tasks* means that multiple transitions have the same label in the original process model. The problem here is that most of the mining techniques treat these duplicate tasks as a single one. For instance, Figure 3 shows a model that also captures the behavior in the log in Table 1 by duplicating the task “Do Theoretical Exam”. *Noise* characterizes low-frequent behavior in the log. It can appear in two situations: event traces were somehow incorrectly logged (for instance, due to temporary system misconfiguration) or event traces reflect exceptional situations. Either way, most of the techniques will try to find a process model that can parse all the traces in the log. However, the presence of noise may hinder the correct mining of the most common behavior.

One of the reasons why the current techniques typically cannot cope with the above mentioned problematic constructs and/or with noisy logs is because their search is based on *local* information in the log. For instance, the α-algorithm (see [5] for details) uses only information about which tasks directly succeed or precede one another in the process instances. As a result, this algorithm does not capture the dependency in non-free-choice constructs. For example, the α-algorithm will never discover the Petri net in Figure 2, for the log in Table 1, because none of the process instances has the sub-trace "Attend Classes Drive Cars, Do Practical Exam Drive Cars" or "Attend Classes Ride Motorbikes, Do Practical Exam Ride Motorbikes". Consequently, the α-algorithm will not link these tasks.

1.3 Genetic Process Mining

To overcome the limitations of the current process mining techniques, our research uses *genetic algorithms* [21] to mine process models. The main motivation is to benefit from the *global* search that is performed by this kind of algorithms.

Genetic algorithms are adaptive search methods that try to mimic the process of evolution. These algorithms start with an initial population of individuals. Every individual is assigned a fitness measure to indicate its quality. In our case, an individual is a possible process model and the fitness is a function that evaluates how well the individual is able to reproduce the behavior in the log. Populations evolve by selecting the fittest individuals and generating new individuals using genetic operators such as *crossover* (combining parts of two or more individuals) and *mutation* (random modification of an individual).

When using genetic algorithms to mine process models, there are three main concerns. The first is to define the *internal representation.* The internal representation defines the search space of a genetic algorithm. The internal representation that we define and explain in this paper supports all the problematic constructs, except for duplicate tasks. The second concern is to define the *fitness measure.* In our case, the fitness measure evaluates the quality of a point (individual or process model) in the search space against the event log. A genetic algorithm searches for individuals whose fitness is maximal. Thus, our fitness measure makes sure that individuals with a maximal fitness can parse all the process instances (traces) in the log and, ideally, not more than those traces. The reason for this is that we aim at discovering a process model that reflects as close as possible the behavior expressed in the event log. If the mined model allows for lots of extra behavior that cannot be derived from the log, it does not give a precise description of what is actually happening. The third concern relates to the *genetic operators* (crossover and mutation) because they should ensure that all points in the search space defined by the internal representation may be reached when the genetic algorithm runs. This paper presents a genetic algorithm that addresses these three concerns.

1.4 Road Map

The remainder of the paper is organized as follows. Section 2 introduces the main definitions of Petri nets that are used in this paper. Section 3 explains the internal representation that we use and defines its semantics by mapping it onto Petri nets. Section 4 presents the genetic algorithm to mine processes that may have arbitrary mixtures of choice and synchronization (i.e., non-free-choice constructs) and invisible tasks. The genetic algorithm is also robust to noisy logs. Section 5 explains the metrics that we have developed to assess the quality of mined models while conducting the experiments. Section 6 discusses the experiments and results. The experiments include synthetic logs. Secion 7 shows the results of applying the genetic algorithm to logs from a municipality in The Netherlands. Section 8 compares the results of the genetic algorithm with the results obtained by two other related process mining techniques. Section 9 discusses the related work. Section 10 contains the conclusions and future work.

2 Preliminaries

This section introduces standard Petri-net notations that are used to explain the semantics of the internal representation of our genetic algorithm.

2.1 Petri Nets

We use a variant of the classic Petri-net model, namely *Place/Transition nets*. For an elaborate introduction to Petri nets, the reader is referred to [18,47,52].

Definition 1 (P/T-nets) [2] *A Place/Transition net, or simply P/T-net, is a tuple* (P, T, F) *where:*

(1) P *is a finite set of* places*,*

[2] In the literature, the class of Petri nets introduced in Definition 2 is sometimes referred to as the class of (unlabeled) *ordinary* P/T-nets to distinguish it from the class of Petri nets that allows more than one arc between a place and a transition, and the class of Petri nets that allows for transition labels.

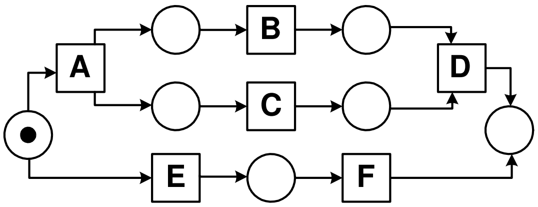


Fig. 4. An example of a Place/Transition net.

(2) T is a finite set of transitions *such that $P \cap T = \emptyset$, and*
(3) $F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs, called the flow relation.

A marked *P/T-net is a pair (N, s), where $N = (P, T, F)$ is a P/T-net and where s is a bag over P denoting the* marking *of the net, i.e. $s \in P \to \mathbb{N}$. The set of all marked P/T-nets is denoted $\mathcal{N}$.*

A marking is a *bag* over the set of places P, i.e., it is a function from P to the natural numbers. We use square brackets for the enumeration of a bag, e.g., $[a^2, b, c^3]$ denotes the bag with two a-s, one b, and three c-s. The sum of two bags $(X + Y)$, the difference $(X - Y)$, the presence of an element in a bag $(a \in X)$, the intersection of two bags $(X \cap Y)$ and the notion of subbags $(X \leq Y)$ are defined in a straightforward way and they can handle a mixture of sets and bags.

Let $N = (P, T, F)$ be a P/T-net. Elements of $P \cup T$ are called *nodes*. A node x is an *input node* of another node y iff there is a directed arc from x to y (i.e., $(x, y) \in F$ or xFy for short). Node x is an *output node* of y iff yFx. For any $x \in P \cup T$, $\overset{N}{\bullet} x = \{y \mid yFx\}$ and $x \overset{N}{\bullet} = \{y \mid xFy\}$; the superscript N may be omitted if clear from the context.

Figure 4 shows a P/T-net consisting of 7 places and 6 transitions. Transition A has one input place and two output places. Transition A is an *AND-split.* Transition D has two input places and one output place. Transition D is an *AND-join.* The black dot in the input place of A and E represents a token. This token denotes the initial marking. The dynamic behavior of such a marked P/T-net is defined by a *firing rule.*

Definition 2 (Firing rule) *Let $N = ((P, T, F), s)$ be a marked P/T-net. Transition $t \in T$ is* enabled, *denoted $(N, s)[t\rangle$, iff $\bullet t \leq s$. The* firing rule $\_\,[\_\rangle\,\_ \subseteq \mathcal{N} \times T \times \mathcal{N}$ *is the smallest relation satisfying for any $(N = (P, T, F), s) \in \mathcal{N}$ and any $t \in T$, $(N, s)[t\rangle \Rightarrow (N, s)\ [t\rangle\ (N, s - \bullet t + t\bullet)$.*

In the marking shown in Figure 4 (i.e., one token in the source place), transitions A and E are enabled. Although both are enabled only one can fire. If transition A fires, a token is removed from its input place and tokens are put in its output places. In the resulting marking, two transitions are enabled: B and C. Note that B and C can be fired concurrently and we assume *inter-*

leaving semantics. In other words, parallel tasks are assumed to be executed in some order.

Definition 3 (Reachable markings) *Let (N, s_0) be a marked P/T-net in $\mathcal{N}$. A marking s is* reachable *from the initial marking s_0 iff there exists a sequence of enabled transitions whose firing leads from s_0 to s. The set of reachable markings of (N, s_0) is denoted $[N, s_0\rangle$.*

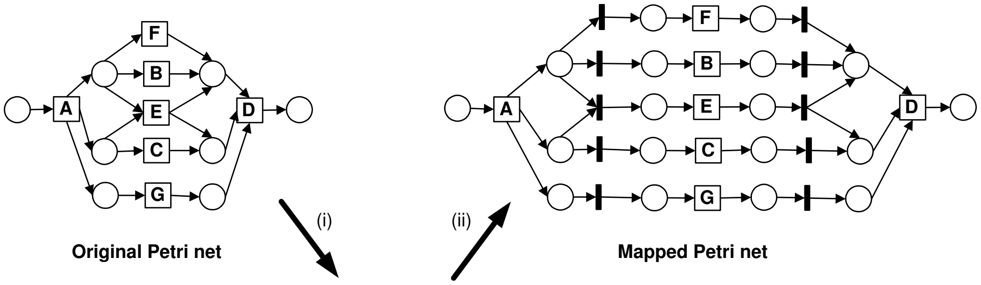
The marked P/T-net shown in Figure 4 has 6 reachable markings. Sometimes it is convenient to know the sequence of transitions that are fired in order to reach some given marking. This paper uses the following notations for sequences. Let A be some alphabet of identifiers. A *sequence of length n*, for some natural number $n \in \mathbb{N}$, over alphabet A is a function $\sigma : \{0, \ldots, n-1\} \rightarrow A$. The sequence of length zero is called the empty sequence and written ε. For the sake of readability, a sequence of positive length is usually written by juxtaposing the function values. For example, a sequence $\sigma = \{(0, a), (1, a), (2, b)\}$, for $a, b \in A$, is written aab. The set of all sequences of arbitrary length over alphabet A is written A^*.

Definition 4 (Firing sequence) *Let (N, s_0) with $N = (P, T, F)$ be a marked P/T net. A sequence $\sigma \in T^*$ is called a* firing sequence *of (N, s_0) if and only if, for some natural number $n \in \mathbb{N}$, there exist markings $s_1, \ldots, s_n$ and transitions $t_1, \ldots, t_n \in T$ such that $\sigma = t_1 \ldots t_n$ and, for all i with $0 \leq i < n$, $(N, s_i)[t_{i+1}\rangle$ and $s_{i+1} = s_i - \bullet t_{i+1} + t_{i+1}\bullet$. (Note that $n = 0$ implies that $\sigma = \varepsilon$ and that ε is a firing sequence of (N, s_0).) Sequence σ is said to be* enabled *in marking s_0, denoted $(N, s_0)[\sigma\rangle$. Firing the sequence σ results in a marking s_n, denoted $(N, s_0)\ [\sigma\rangle\ (N, s_n)$.*

For the marked Petri net shown in Figure 4, some possible firing sequences are $ABCD$, $ACBD$ and AE. Note that, for these firing sequences, the resulting marking has a single token and this token is in the output place of transitions D and F.

3 Internal Representation and Semantics

When defining the internal representation to be used by our genetic algorithm, the main requirement was that this representation should express the dependencies between the tasks in the log. In other words, the model should clearly express which tasks would enable the execution of other tasks. Additionally, it would be nice if the internal representation would be compatible with a formalism to which analysis techniques and tools exist. This way, these techniques could also be applied to the discovered models. Thus, one option would be to directly represent the individual (or process model) as a Petri net [18,47]. How-



ACTIVITY	*I(ACTIVITY)*	*O(ACTIVITY)*
A	{}	{{F,B,E},{E,C},{G}}
B	{{A}}	{{D}}
C	{{A}}	{{D}}
D	{{F,B,E},{E,C},{G}}	{}
E	{{A}}	{{D}}
F	{{A}}	{{D}}
G	{{A}}	{{D}}

Compact representation of the causal matrix

Fig. 5. Mapping of a PN with more than one place between two tasks (or transitions).

ever, such a representation would require determining the number of places in every individual and this is not the core concern. It is more important to show the dependencies between the tasks and the semantics of the split/join tasks. Therefore, we defined an internal representation that is as expressive as Petri nets (from the task dependency perspective) but that only focuses on the tasks. This representation is called *causal matrix.* Figure 5 illustrates in (i) the causal matrix that expresses the same task dependencies that are in the "original Petri net". The causal matrix shows which tasks enable the execution of other tasks via the matching of *input* (I) and *output* (O) condition functions. The sets returned by the condition functions I and O have *subsets* that contain the tasks in the model. Tasks in a same subset have an XOR-split/join relation. Sets in different subsets have an AND-split/join relation. Thus, every I and O set expresses a conjunction of exclusive disjunctions. Additionally, a task may appear in more than one subset in a same set. As an example, for task D in the original Petri net in Figure 5 the causal matrix states that $I(D) = \{\{F, B, E\}, \{E, C\}, \{G\}\}$ because D is enabled by an AND-join construct that has 3 places. From top to bottom, the first place has a token whenever F *or* B *or* E fires. The second place, whenever E *or* C fires. The third place, whenever G fires. Similarly, the causal matrix has $O(D) = \{\}$ because D is executed last in the model. The following definition formally defines these notions.

Definition 5 (Causal Matrix) *A Causal Matrix is a tuple* $CM = (A, C, I, O)$, *where*

- A *is a finite set of activities,*
- $C \subseteq A \times A$ *is the causality relation,*

- $I : A \rightarrow \mathcal{P}(\mathcal{P}(A))$ *is the input condition function,*[3]
- $O : A \rightarrow \mathcal{P}(\mathcal{P}(A))$ *is the output condition function,*

such that

- $C = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup I(a_2)\}$,[4]
- $C = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup O(a_1)\}$,
- $C \cup \{(a_o, a_i) \in A \times A \mid a_o \overset{C}{\bullet} = \emptyset \wedge \overset{C}{\bullet} a_i = \emptyset\}$ *is a strongly connected graph,*

The set of all causal matrices is denoted by $\mathcal{CM}$, and a bag of causal matrices is denoted by $\mathcal{CM}[]$.

Any Petri net without duplicate tasks and without more than one place with the same input tasks and the same output tasks can be mapped to a causal matrix. Definition 6 formalizes such a mapping. The main idea is that there is a causal relation C between any two tasks t and t' whenever at least one of the *output* places of t is an *input* place of t'. Additionally, the I and O condition functions are based on the input and output places of the tasks. This is a natural way of mapping because the input and output places of Petri nets actually reflect the conjunction of disjunctions that these sets express.

Definition 6 **($\Pi_{PN \rightarrow CM}$)** *Let $PN = (P, T, F)$ be a Petri net. The mapping of PN is a tuple $\Pi_{PN \rightarrow CM}(PN) = (A, C, I, O)$, where*

- $A = T$,
- $C = \{(t_1, t_2) \in T \times T \mid t_1 \bullet \cap \bullet t_2 \neq \emptyset\}$,
- $I \in T \rightarrow \mathcal{P}(\mathcal{P}(T))$ *such that* $\forall_{t \in T}\ I(t) = \{\bullet p \mid p \in \bullet t\}$,
- $O \in T \rightarrow \mathcal{P}(\mathcal{P}(T))$ *such that* $\forall_{t \in T}\ O(t) = \{p \bullet \mid p \in t \bullet\}$.

The semantics of the causal matrix can be easily understood by mapping them back to Petri nets. This mapping is formalized in Definition 7. Conceptually, the causal matrix behaves as a Petri net that contains visible and invisible tasks. For instance, see Figure 5. This figure shows (i) the mapping of a Petri net to a causal matrix and (ii) the mapping from the causal matrix to a Petri net. The firing rule for the mapped Petri net is very similar to the firing rule of Petri nets in general (cf. Definition 2). The only difference concerns the invisible tasks. Enabled invisible tasks can only fire if their firing enables a visible task. Similarly, a visible task is enabled if all of its input places have tokens or if there exits *a set of* invisible tasks that are enabled and whose firing will lead to the enabling of the visible task. Conceptually, the causal matrix keeps track of the distribution of tokens at a marking in the output places of the visible tasks. The invisible tasks can be seen as “channels” or “pipes” that are only used when a visible task needs to fire. Every causal

[3] $\mathcal{P}(A)$ denotes the powerset of some set A.

[4] $\bigcup I(a_2)$ is the union of the sets in set $I(a_2)$.

matrix starts with a token at the start place. Finally, we point out that, in Figure 5, although the mapped Petri net does not have the same *structure* of the original Petri net, these two nets are *behaviorally* equivalent. In other words, given that these two nets initially have a single token and this token is at the start place (i.e., the input place of A), the set of traces the two nets can generate is the same.

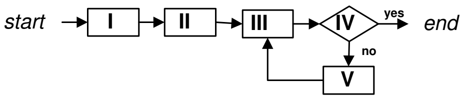
Definition 7 **($\Pi^N_{CM \rightarrow PN}$)** *Let $CM = (A, C, I, O)$ be a causal matrix. The naive Petri net mapping of CM is a tuple $\Pi^N_{CM \rightarrow PN} = (P, T, F)$, where*

- $P = \{i, o\} \cup \{i_{t,s} \mid t \in A \ \wedge \ s \in I(t)\} \cup \{o_{t,s} \mid t \in A \ \wedge \ s \in O(t)\}$,
- $T = A \cup \{m_{t_1,t_2} \mid (t_1, t_2) \in C\}$,
- $F = \{(i, t) \mid t \in A \wedge \overset{C}{\bullet} t = \emptyset\} \cup \{(t, o) \mid t \in A \ \wedge \ t \overset{C}{\bullet} = \emptyset\} \cup \{(i_{t,s}, t) \mid t \in A \ \wedge \ s \in I(t)\} \cup \{(t, o_{t,s}) \mid t \in A \ \wedge \ s \in O(t)\} \cup \{(o_{t_1,s}, m_{t_1,t_2}) \mid (t_1, t_2) \in C \wedge s \in O(t_1) \wedge t_2 \in s\} \cup \{(m_{t_1,t_2}, i_{t_2,s}) \mid (t_1, t_2) \in C \wedge s \in I(t_2) \wedge t_1 \in s\}$.

Definition 7 shows a rather naive approach to generate the mapped Petri net shown in Figure 5. However, as shown in [16], there are special situations in which more sophisticated mappings are possible.

4 Genetic Algorithm

In this section we describe the main steps of our genetic algorithm. Figure 6 shows how they are related.



Step	Description
I	*Read event log*
II	*Build the initial population*
III	*Calculate fitness of the individuals in the population*
IV	*Stop and return the fittest individuals?*
V	*Create next population – use elitism and genetic operators*

Fig. 6. Main steps of our genetic algorithm.

4.1 Initial Population

The initial population is randomly built by the genetic algorithm. As explained in Section 3, individuals are causal matrices. When building the initial population, we ensure that the individuals comply with Definition 5. Given a log, all individuals in any population of the genetic algorithm have the same set of activities (or tasks) A. This set contains the tasks that appear in the log.

The setting of the causality relation C can be done via a completely random approach or a heuristic one. The random approach uses 50% probability for establishing (or not) a causality relation between two task in A. The heuristic approach uses the information in the log to determine the probability that two tasks are going to have a causality relation set. In a nutshell, the heuristics works as follows: the more often a task t_1 is directly followed by a task t_2 (i.e. the subtrace "t_1, t_2" appears in traces in the log), the higher the probability that individuals are built with a causality relation from t_1 to t_2 (i.e., $(t_1, t_2) \in C$). This heuristic way of building an individual is based on the work presented in [60]. Subsection 4.1.1 has more details about the heuristic approach. Once the causality relations of an individual are determined, the condition functions I and O are randomly built. This is done by setting a maximum size n for any input or output condition function set of a task t in the initial population[5]. Every task t_1 that causally precedes a task t_2, i.e. $(t_1, t_2) \in C$, is randomly inserted in *one or more* subsets of the input condition function of t_2. A similar process is done to set the output condition function of a task[6]. In our case, we set the number of distinct tasks in the log as the maximum size for any input/output condition function set in the initial population[7]. As a result, the initial population can have any individual in the search space defined by a set of activities A, and that satisfy the constraints for the size of the input/output condition function sets. Note that the higher the amount of tasks that a log contains, the bigger this search space. Finally, we emphasize that no further limitations to the input/output condition functions sets are made in the other steps of the genetic algorithm. Therefore, during the "Step V" in the Figure 6, these sets can increase or shrink as the population evolves.

4.1.1 Heuristics to Build the Causality Relation of a Causal Matrix

When applying a genetic algorithm to a domain, it is common practice to "give a hand" to the genetic algorithm by using well-know heuristics (in this domain) to build the initial population [21]. Studies show that the use of heuristics often does not alter the end result (if the genetic algorithm runs for infinite amount of time), but it may speed the early stages of the evolution. The GAs that use heuristics are called *hybrid genetic algorithms*.

In our specific domain - process mining - some heuristics have proven to give reasonable solutions when used to mine event logs. These heuristics are mostly based on local information in the event-log. Due to its similarities to other related work, we use the heuristics in [60] to guide the setting of the causality

[5] Formally: $\forall_{t \in A}[|I(t)| \leq n \ \wedge \ |O(t)| \leq n]$.

[6] Formally: $\forall_{t_1,t_2 \in A,(t_1,t_2) \in C}[\exists i \in I(t_2) : t_1 \in i]$ and $\forall_{t_1,t_2 \in A,(t_1,t_2) \in C}[\exists o \in O(t_1) : t_2 \in o]$.

[7] Formally: $\forall_{t \in A}[|I(t)| \leq |A| \wedge |O(t)| \leq |A|]$.

relations in the individuals of the initial population. These heuristics are based on the *dependency measure*. To define this measure, we first need to formalize the notion of an *event log*.

Definition 8 (Event Trace, Event Log) *Let* T *be a set of tasks.* $\sigma \in T^*$ *is an* event trace *and* $L : T^* \rightarrow \mathbb{N}$ *is an* event log. *For any* $\sigma \in dom(L)$, $L(\sigma)$ *is the number of occurrences of* σ. *The set of all event logs is denotes by* $\mathcal{L}$.

Note that we use $dom(f)$ and $rng(f)$ to respectively denote the *domain* and *range* of a function f. Furthermore, we use the notation $\sigma \in L$ to denote $\sigma \in dom(L) \wedge L(\sigma) \geq 1$. For example, assume a log $L = [abcd, acbd, abcd]$ for the net in Figure 4. Then, we have that $L(abcd) = 2$, $L(acbd) = 1$ and $L(ab) = 0$.

The dependency measure basically indicates how strongly a task depends (or is caused) by another task. The more often a task t_1 *directly* precedes another task t_2 in the log, and the less often t_2 *directly* precedes t_1, the stronger is the dependency between t_1 and t_2. In other words, the more likely it is that t_1 is a *cause* to t_2. The dependency measure is given in Definition 9. The notation used in this definition is as follows. $l2l : T \times T \times \mathcal{L} \rightarrow \mathbb{N}$ is a function that detects length-two loops. *l2l* gives the number of times that the substring "$t_1 t_2 t_1$" occurs in the log L. $follows : T \times T \times \mathcal{L} \rightarrow \mathbb{N}$ is a function that returns the number of times that a task is directly followed by another one. That is, how often the substring "$t_1 t_2$" occurs in the log L.

Definition 9 (Dependency Measure - D **)** *Let* L *be an event log. Let* T *be the set of tasks in* L. *Let* t_1 *and* t_2 *be two tasks in* T. *The dependency measure* $D : T \times T \times \mathcal{L} \rightarrow \mathbb{R}$ *is a function defined as:*

$$D(t_1, t_2, L) = \begin{cases} \frac{l2l(t_1,t_2,L)+l2l(t_2,t_1,L)}{l2l(t_1,t_2,L)+l2l(t_2,t_1,L)+1} & \text{if } t_1 \neq t_2 \text{ and } l2l(t_1,t_2,L) > 0, \\ \\ \frac{follows(t_1,t_2,L)-follows(t_2,t_1,L)}{follows(t_1,t_2,L)+follows(t_2,t_1,L)+1} & \text{if } t_1 \neq t_2 \text{ and } l2l(t_1,t_2,L) = 0, \\ \\ \frac{follows(t_1,t_2,L)}{follows(t_1,t_2,L)+1} & \text{if } t_1 = t_2. \end{cases}$$

Observe that the dependency relation distinguishes between tasks in short loops (length-one and length-two loops) and tasks in parallel. Moreover, the "+1" in the denominator is used to benefit more frequent observations over less frequent ones. For instance, if a length-one-loop "tt" happens only once in the log L, the dependency measure $D(t, t, L) = 0.5$. However, if this same length-one-loop would occur a hundred times in the log, $D(t, t, L) = 0.99$. Thus, the more often a substring (or pattern) happens in the log, the stronger the dependency measure.

Once the dependency relations are set for the input event log, the genetic algorithm uses it to randomly build the causality relations for every individual in the initial population. The pseudo-code for this procedure is the following:

Pseudo-code:
input: An event-log L, a power value p, the dependency function D.
output: A causality relation C.

```
(1) $T \longleftarrow$ set of tasks in $L$.
(2) $C \longleftarrow \emptyset$.
(3) FOR every tuple $(t_1, t_2)$ in $T \times T$ do:
    (a) Randomly select a number $r$ between 0 (inclusive) and 1.0 (exclusive).
    (b) IF $r < D(t_1, t_2, L)^p$ then:
        (i) $C \longleftarrow C \cup \{(t_1, t_2)\}$.
(4) Return the causality relation $C$.
```

Note that we use a power value p to control the "influence" of the dependency measure in the probability of setting a causality relation. Higher values for p lead to the inference of fewer causality relations among the tasks in the event log, and vice-versa.

4.2 Fitness Calculation

As discussed in Section 1, process mining aims at discovering a process model from an event log. This mined process model should give a good insight about what the behavior in the log is. In other words, the mined process model should be *complete and precise from a behavioral perspective.* A process model is complete when it can parse (or reproduce) all the event traces in the log. A process model is precise when it cannot parse more than the traces in the log. The requirement that the mined model should also be precise is important because different models are able to parse all event traces and these models may allow for extra behavior that does not belong to the log. To illustrate this we consider the nets shown in Figure 7. These models can also parse the traces in Table 1, but they allow for extra behavior. For instance, both models allow for the applicant to take the exam before attending to classes. The fitness function guides the search process of the genetic algorithm. Thus, the fitness of an individual is assessed by benefiting the individuals that can parse more event traces in the log (the "completeness" requirement) and by punishing the individuals that allow for more extra behavior than the one expressed in the log (the "preciseness" requirement).

To facilitate the explanation of our fitness measure, we divide it into three parts. First, we discuss in Subsection 4.2.1 how we defined the part of the fitness measure that guides the genetic algorithm towards individuals that are

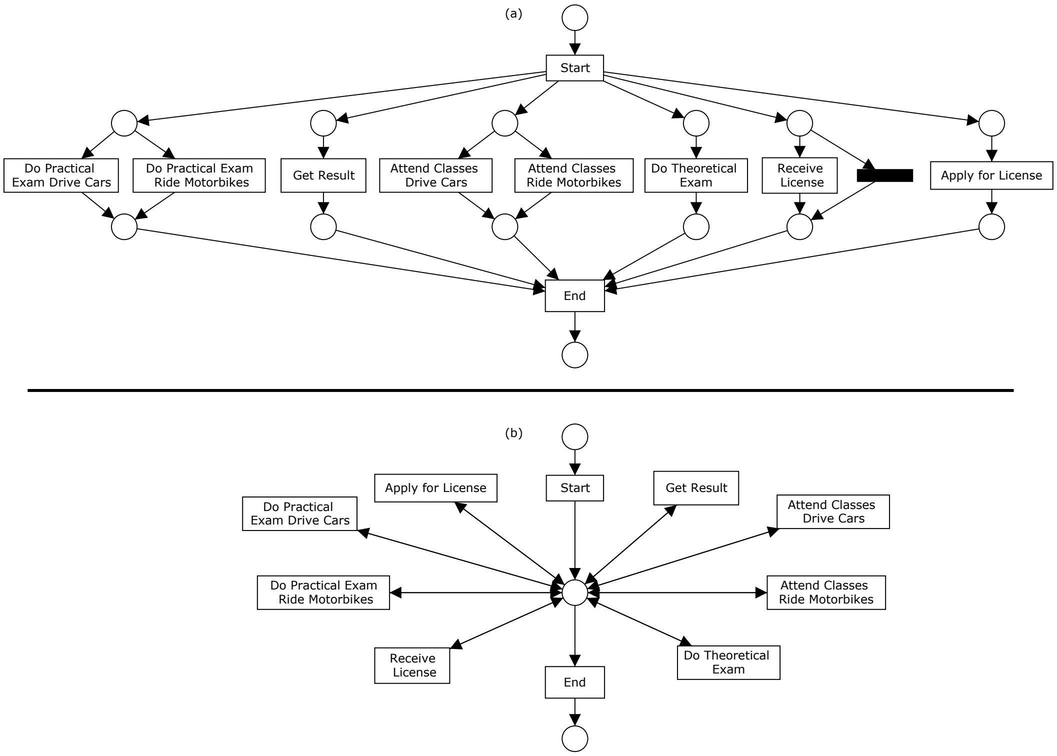

(a)
Start
Do Practical Exam Drive Cars
Do Practical Exam Ride Motorbikes
Get Result
Attend Classes Drive Cars
Attend Classes Ride Motorbikes
Do Theoretical Exam
Receive License
Apply for License
End
(b)
Apply for License
Start
Get Result
Do Practical Exam Drive Cars
Attend Classes Drive Cars
Do Practical Exam Ride Motorbikes
Attend Classes Ride Motorbikes
Receive License
End
Do Theoretical Exam

Fig. 7. Example of nets that can also reproduce the behavior for the log in Table 1. The problem here is that these nets allow for extra behavior that is not in the log.

more complete. Second, we show in Subsection 4.2.2 how we defined the part of the fitness measure that benefits individuals that are more precise. Finally, we show in Subsection 4.2.3 the fitness measure that our genetic algorithm is using. This fitness measure combines the partial fitness measures that are presented in the subsections 4.2.1 and 4.2.2.

4.2.1 The "Completeness" Requirement

The "completeness" requirement of our fitness measure is based on the parsing of event traces by individuals. For a noise-free log, the perfect individual should have fitness 1. This means that this individual could parse all the traces in the log. Therefore, a natural fitness for an individual to a given log seems to be the number of properly parsed event traces[8] divided by the total number of event traces. However, this fitness measure is too coarse because it does not give an indication about (i) how many parts of an individual are correct when the individual does not properly parse an event trace and

[8] An event trace is properly parsed by an individual if, for an initial marking that contains a single token and this token is at the start place of the mapped Petri net for this individual, after firing the visible tasks in the order in which they appear in the event trace, the end place is the only one to be marked and it has a single token.

(ii) the semantics of the split/join tasks. For instance, if a net has an AND-split instead of an XOR-split, it may happen that all tasks in a trace can be replayed by this net, but this net does not proper complete for this trace because tokens remain at some of the output places of the AND-split task. So, we defined a more elaborate fitness function: when the task to be parsed is not enabled, the problems (e.g. number of missing tokens to enable this task) are registered and the parsing proceeds as if this task would be enabled. This *continuous* parsing semantics is more robust because it gives a better indication of how many tasks do or do not have problems during the parsing of a trace. The partial fitness function that tackles the "completeness" requirement is in Definition 10. The notation used in this definition is as follows. $\mathit{allParsedActivities}(L, CM)$ gives the total number of tasks in the event log L that could be parsed without problems by the causal matrix (or individual) CM. $\mathit{numActivitiesLog}(L)$ gives the number of tasks in L. $\mathit{allMissingTokens}(L, CM)$ indicates the number of missing tokens in all event traces. $\mathit{allExtraTokensLeftBehind}(L, CM)$ indicates the number of tokens that were not consumed after the parsing has stopped plus the number of tokens of the end place minus 1 (because of proper completion). $\mathit{numTracesLog}(L)$ indicates the number of traces in L. $\mathit{numTracesMissingTokens}(L, CM)$ and $\mathit{numTracesExtraTokensLeftBehind}(L, CM)$ respectively indicate the number of traces in which tokens were missing and tokens were left behind during the parsing.

Definition 10 (Partial Fitness - PF_{complete} **)** *Let L be a non-empty event log. Let CM be a causal matrix. Then the partial fitness $PF_{\mathit{complete}} : \mathcal{L} \times \mathcal{CM} \rightarrow (-\infty, 1]$ is a function defined as:*

$$PF_{\mathit{complete}}(L, CM) = \frac{\mathit{allParsedActivities}(L, CM) - \mathit{punishment}}{\mathit{numActivitiesLog}(L)}$$

where

$$\mathit{punishment} =$$

$$\frac{\mathit{allMissingTokens}(L, CM)}{\mathit{numTracesLog}(L) - \mathit{numTracesMissingTokens}(L, CM) + 1} +$$

$$\frac{\mathit{allExtraTokensLeftBehind}(L, CM)}{\mathit{numTracesLog}(L) - \mathit{numTracesExtraTokensLeftBehind}(L, CM) + 1}$$

The partial fitness PF_{complete} gives a more detailed indication about how fit an individual is to a given log. The function *allMissingTokens* penalizes (i) nets with XOR-split where it should be an AND-split and (ii) nets with an AND-join where it should be an XOR-join. Similarly, the function *allExtraTokensLeftBehind* penalizes (i) nets with AND-split where it should be an XOR-split and (ii) nets with an XOR-join where it should be an AND-join. Note that we weigh the impact of the *allMissingTokens* and *allExtraTokensLeftBehind* functions by respectively dividing them by the number of event traces minus the number of event traces with missing and left-behind tokens. The main idea is to promote individuals that correctly parse the more frequent behavior in the log. Additionally, if two individuals have the same *punishment* value, the one that can parse more tasks has a better fitness because its missing and left-behind tokens impact fewer tasks. This may indicate that this individual has more correct I and O condition functions than incorrect ones. In other words, this individual is a better candidate to produce offsprings for the next population (see Subsection 4.4).

4.2.2 The "Preciseness" requirement

The "preciseness" requirement is based on discovering how much extra behavior an individual allows for. To define a fitness measure to punish models that express more than it is in the log is especially difficult because we do not have negative examples to guide our search. Note that the event logs show the allowed (positive) behavior, but they do not express the forbidden (negative) one.

One possible solution to punish an individual that allows for undesirable behavior could be to build the *coverability graph* [47] of the mapped Petri net for this individual and check the fraction of event traces this individual can generate that are not in the log. The traces that express different paths of execution for parallelism are not considered as extra behavior. The main idea in this approach is to punish the individual for every extra event trace it generates. Unfortunately, building the coverability graph is not very practical and it is unrealistic to assume that all possible behavior is present in the log.

Because proving that a certain individual is precise is not practical, we use a simpler solution to guide our genetic algorithm towards solutions that have "less extra behavior". We check, for every marking, the *number of visible tasks that are enabled.* Individuals that allow for extra behavior tend to have more enabled tasks than individuals that do not. For instance, the nets in Figure 7 have more enabled tasks in most reachable markings than the net in Figure 2. The main idea in this approach is to benefit individuals that have a smaller amount of enabled tasks during the parsing of the log. This is the measure we use to define our second partial fitness function $PF_{precise}$ that is presented in Definition 11. The notation used in this definition is as follows. *allEnabledActivities*(L, CM) indicates the number of activities that were enabled during the parsing of the log L by the causal matrix (or individual) CM. *allEnabledActivities*$(L, CM[])$ apply *allEnabledActivities*(L, CM) (see notation for Definition 10) to every element in the bag of causal matrices (or population) $CM[]$. The function $max(allEnabledActivities(L, CM[]))$ returns the *maximum* value of the amount of enabled tasks that individuals in the given population ($CM[]$) had while parsing the log (L).

Definition 11 (Partial Fitness - $PF_{precise}$**)** *Let* L *be a non-empty event log. Let* CM *be a causal matrix. Let* $CM[]$ *be a bag of causal matrices that contains* CM*. The partial fitness* $PF_{precise} : \mathcal{L} \times \mathcal{CM} \times \mathcal{CM}[] \rightarrow [0, 1]$ *is a function defined as*

$$PF_{precise}(L, CM, CM[]) = \frac{allEnabledActivities(L, CM)}{max(allEnabledActivities(L, CM[]))}$$

The partial fitness $PF_{precise}$ gives an indication of how much extra behavior an individual allows for *in comparison to* other individuals in the same population. The smaller the $PF_{precise}$ of an individual is, the better. This way we avoid over-generalizations.

4.2.3 Fitness - Combining the "Completeness" and "Preciseness" Requirements

While defining the fitness measure, we decided that the "completeness" requirement should be more relevant than the "preciseness" one. The reason is that we are only interested in precise models that are also complete. The resulting fitness is defined as follows.

Definition 12 (Fitness - F**)** *Let* L *be a non-empty event log. Let* CM *be a causal matrix. Let* $CM[]$ *be a bag of causal matrices that contains* CM*. Let* $PF_{complete}$ *and* $PF_{precise}$ *be the respective partial fitness functions given in definitions 10 and 11. Let* κ *be a real number greater than* 0 *and smaller or equal to* 1 *(i.e.,* $\kappa \in (0, 1]$*). Then the fitness* $F : \mathcal{L} \times \mathcal{CM} \times \mathcal{CM}[] \rightarrow (-\infty, 1)$ *is a function defined as*

$$F(L, CM, CM[]) = PF_{complete}(L, CM) - \kappa * PF_{precise}(L, CM, CM[])$$

The fitness F weighs (by κ) the punishment for extra behavior. Thus, if a set of individuals can parse all the traces in the log, the one that allows for less extra behavior will have a higher fitness value. For instance, assume a population with the corresponding individual for the net in Figure 2 and the corresponding individuals for the nets in Figure 7. If we calculate the fitness F of these three individuals with respect to the log in Table 1, the individual in Figure 2 will have the highest fitness value among the three and the individual in Figure 7(b), the lowest fitness value.

4.3 Stop Criteria

The mining algorithm stops when (i) it computes n generations, where n is the maximum number of generations that is allowed; or (ii) the fittest individual has not changed for $n/2$ generations in a row.

4.4 Genetic Operators

We use *elitism*, *crossover* and *mutation* to build the individuals of the next generation. A percentage of the best individuals (the *elite*) is directly copied to the next population. The other individuals in the population are generated via crossover and mutation. Two parents produce two offsprings. To select one parent, a *tournament* is played in which five individuals in the population are randomly drawn and the fittest one always wins. The crossover and mutation operator are explained respectively in subsections 4.4.1 and 4.4.2.

4.4.1 Crossover

Crossover is a genetic operator that aims at recombining existing material in the current population. In our case, this material is the current causality relations (cf. Definition 5) in the population. Thus, the crossover operator used by our genetic algorithm should allow for the complete search of the space defined by the existing causality relation in a population. Given a set of causality relations, the search space contains all the individuals that can be created by any combination of a subset of the causality relations in the population. Thus, our crossover operator allows an individual to: lose tasks from the subsets in its I/O condition functions (but not necessarily causality relations because a same task may be in more than one subset of an I/O condition function), add tasks to the subsets in its I/O condition functions (again, not necessarily causality relations), exchange causality relations with other individuals, incorporate causality relations that are in the population but are not in the individual, lose causality relations, decrease the number of subsets in its I/O condition functions, *and/or* increase the number of subsets in its I/O condition functions. The *crossover rate* determines the probability that two parents undergo crossover. The crossover point of two parents is a randomly chosen task. The pseudo-code for the crossover operator is as follows:

Pseudo-code:
input: Two parents ($parent_1$ and $parent_2$), crossover rate.
output: Two possibly recombined offsprings ($offspring_1$ and $offspring_2$).

```
(1) $offspring_1 \longleftarrow parent_1$ and $offspring_2 \longleftarrow parent_2$.
```

```
(2) With probability “crossover rate” do:
    (a) Randomly select a task $t$ to be the crossover point of the offsprings.
    (b) Randomly select a swap point $sp_1$ for $I_1(t)$ [9]. The swap point goes
        from position 0 to $n-1$, where $n$ is the number of subsets in the
        condition function $I_1(t)$.
    (c) Randomly select a swap point $sp_2$ for $I_2(t)$.
    (d) $remainingSet_1(t)$ equals subsets in $I_1(t)$ that are between position 0
        and $sp_1$ (exclusive).
    (e) $swapSet_1(t)$ equals subsets in $I_1(t)$ whose position equals or bigger
        than $sp_1$.
    (f) Repeat steps 2d and 2e but respectively use $remainingSet_2(t)$, $I_2(t)$,
        $sp_2$ and $swapSet_2(t)$ instead of $remainingSet_1(t)$, $I_1(t)$, $sp_1$ and
        $swapSet_1(t)$.
    (g) FOR every subset $S_2$ in $swapSet_2(t)$ do:
        (i) With equal probability perform one of the following steps:
            (A) Add $S_2$ as a new subset in $remainingSet_1(t)$.
            (B) Join $S_2$ with an existing subset $X_1$ in $remainingSet_1(t)$.
            (C) Select a subset $X_1$ in $remainingSet_1(t)$, remove the elements
                of $X_1$ that are also in $S_2$ and add $S_2$ to $remaining-Set_1(t)$.
    (h) Repeat Step 2g but respectively use $S_1$, $swapSet_1(t)$, $X_2$ and $remain-ingSet_2(t)$
        instead of $S_2$, $swapSet_2(t)$, $X_1$ and $remainingSet_1(t)$.
    (i) $I_1(t) \longleftarrow remainingSet_1(t)$ and $I_2(t) \longleftarrow remainingSet_2(t)$.
    (j) Repeat steps 2b to 2h but use $O(t)$ instead of the $I(t)$.
    (k) Update the related tasks to $t$.
(3) Return $offspring_1$ and $offspring_2$.
```

Note that, after crossover, the number of causality relations for the whole population remains constant, but how these relations appear in the offsprings may be different from the parents. Moreover, the offsprings may be different even when both parents are equal. For instance, consider the situation in which the crossover operator receives as input two parents that are equal to the causal matrix in Figure 5. Assume that (i) the crossover point is the task D, (ii) we are doing crossover over the input condition function $I(D) = \{\{F, B, E\}, \{E, C\}, \{G\}\}$, and (iii) the swap points are $sp_1 = 1$ and $sp_2 = 2$. Then, we have that the $remainingSet_1(D) = \{\{F, B, E\}\}$, the $swapSet_1(D) = \{\{E, C\}, \{G\}\}$, the $remainingSet_2(D) = \{\{F, B, E\}, \{E, C\}\}$, the $swapSet_2(D) = \{\{G\}\}$. Let us first crossover the subsets in the $swapSet_2(D)$ with the $remainingSet_1(D)$. During the crossover, the genetic algorithm randomly chooses to merge the subset $S_2 = \{G\}$ in the $swapSet_2(D)$ with the existing subset $X_1 = \{F, B, E\}$. In a similar way, while swapping the subsets

[9] We use the notation $I_j(t)$ to get the subset returned by the input condition function I of task t in individual j. In this pseudo-code, the individuals are the offsprings (*offspring*$_1$ and *offspring*$_2$) and $j \in \{1, 2\}$.

in *swapSet*$_1(D)$ with the *remainingSet*$_2(D)$, the algorithm randomly chooses (i) to insert the subset $S_1 = \{E, C\}$ and remove task E from the subset $X_2 = \{F, B, E\}$, and (ii) to insert the subset $S_1 = \{G\}$ as a new subset in the *remainingSet*$_2(D)$. The result is that $I_1(D) = \{\{F, B, E, G\}\}$ and $I_2(D) = \{\{F, B\}, \{E, C\}, \{G\}\}$. The output condition functions $O_1(D)$ and $O_2(D)$ do not change after the crossover operator because the task D does not have any output task. After the crossover, the mutation operator takes place.

4.4.2 Mutation

The mutation operator aims at inserting new material in the current population. In our case, this means that the mutation operator may change the existing causality relations of a population. Thus, our mutation operator performs one of the following actions to the I/O condition functions of a task in an individual: (i) randomly choose a subset and add a task (in A) to this subset, (ii) randomly choose a subset and remove a task out of this subset, *or* (iii) randomly redistribute the elements in the subsets of I/O into new subsets. For example, consider the input condition function of task D in Figure 5. $I(D) = \{\{F, B, E\}, \{E, C\}, \{G\}\}$ can be mutated to (i) $\{\{F, B, E\}, \{E, C\}, \{G, D\}\}$ if task D is added to the subset $\{G\}$, (ii) $\{\{F, B, E\}, \{C\}, \{G\}\}$ if task E is removed from the subset $\{E, C\}$, or (iii) $\{\{F\}, \{E, C, B\}, \{G\}, \{E\}\}$ if the elements in the original $I(D)$ are randomly redistributed in a randomly chosen number of new subsets. Every task in an offspring may undergo mutation with the probability determined by the *mutation rate*. The pseudo-code for the mutation operator is as follows:

Pseudo-code:
input: An individual, mutation rate.
output: A possibly mutated individual.

```
(1) For every task $t$ in the individual do:
    (a) With probability mutation rate do one of the following operations for
        the condition function $I(t)$:
        (i) Select a subset $X$ in $I(t)$ and add a task $t'$ to $X$, where $t'$ belongs
            to the set of tasks in the individual.
        (ii) Select a subset $X$ in $I(t)$ and remove a task $t'$ from $X$, where
            $t'$ belongs to $X$. If $X$ is empty after $t'$ removal, exclude $X$ from
            $I(t)$.
        (iii) Redistribute the elements in $I(t)$. [10]
    (b) Repeat Step 1a, but use the condition function $O(t)$ instead of $I(t)$.
```

[10] Details about this step: (i) Get a list with the elements of $I(t)$; (ii) Create n sets (n is the number of elements in $I(t)$) and randomly distribute them in the n sets; and (iii) Filter out the non-empty sets. These non-empty sets are now the subsets of $I(t)$.

(c) Update the related tasks to t.

As the reader may already have noticed, both the crossover and the mutation operators perform a repairing operation at the end of their executions. The "update the related tasks" operation makes sure that the individual is still compliant with the Definition 5 after undergoing crossover and/or mutation.

5 Analysis Metrics

The genetic algorithm searches for models that are *complete* and *precise* (see Subsection 4.2). Therefore, when evaluating the results of our experiments, we should check if the mined models are indeed complete and precise. At first sight, the natural way to check for this seemed to be to compare the causal matrix of the original model (the one that was simulated to created the synthetic event logs) with the causal matrix of the individual that was mined by the genetic algorithm. However, this is not a good evaluation criterion because there are different ways to model the exact behavior expressed in a log. For instance, consider the net in Figure 8. This net produces exactly the same behavior as the one in Figure 2. However, their causal matrices are different. Furthermore, even when the mined models are not complete and/or precise, we should be able to assess how much correct material they contain. This is important because we do not let the genetic algorithm run for an "infinite" amount of time. Thus, even when the mined model is not complete and precise, it is important to know if the genetic algorithm is going in the right direction.

In our experiments, we have three elements: (i) the *original model* that is used to build the synthetic event log, (ii) the synthetic *event log* itself, and (iii) the *mined model* (or individual). Thus, to analyse our results, we have defined metrics that are based on two or more of these elements.

Checking for Completeness

To check for completeness, we only need the event log and the mined model. Recall that a model is complete when it can parse all the traces in the log without having missing tokens or tokens left behind. So, completeness can be verified by calculating the partial fitness $PF_{complete}$ (see Definition 10) for the event log and the mined model. Whenever $PF_{complete} = 1$, the mined model is complete. Moreover, even when the mined model has $PF_{complete} < 1$, this measure gives an indication of the quality of the mined model with respect to completeness.

Checking for Preciseness

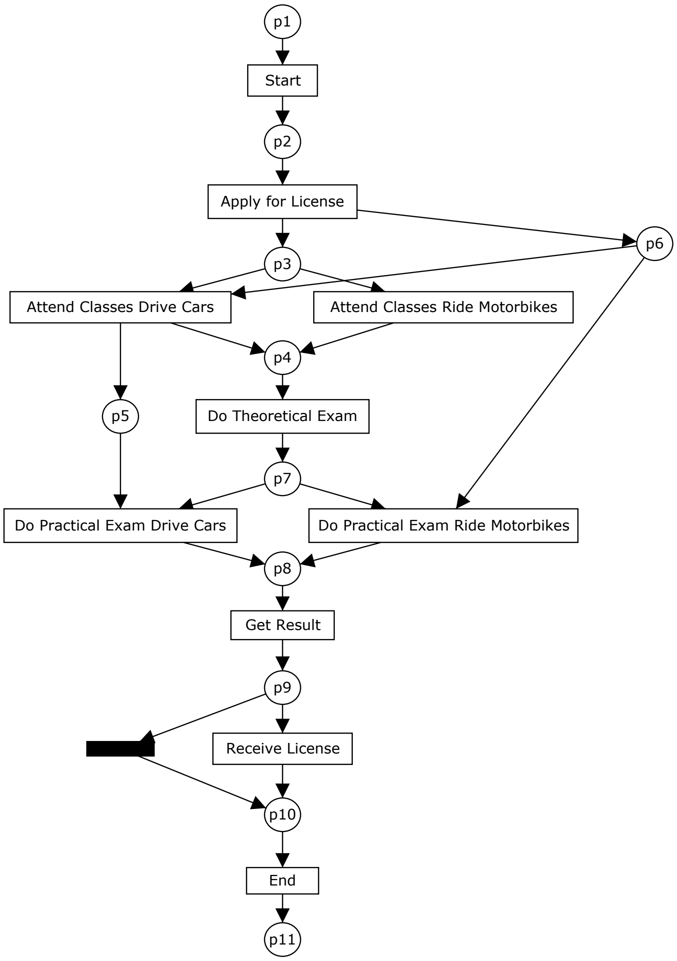

p1
Start
p2
Apply for License
p6
p3
Attend Classes Drive Cars
Attend Classes Ride Motorbikes
p4
p5
Do Theoretical Exam
p7
Do Practical Exam Drive Cars
Do Practical Exam Ride Motorbikes
p8
Get Result
p9
Receive License
p10
End
p11

Fig. 8. Other mined net for the log in Table 1. Note that this net is behaviorally equivalent to the net in Figure 2, although they are structurally different. Note that the place "p6" has different input and output tasks in the two nets.

To check for preciseness, we need the original model, the event log and the mined model[11] . The main reason why we could not define metrics only based on the event log and the mined model, or the original model and the mined model, is because it is unrealistic to assume that the event log has all the possible traces that the original model can generate. In other words, it is unrealistic to assume that the log contains all possible event traces. Recall that a model is precise when it does not allow for more behavior than the one expressed in the log. Thus, if the log would be exhaustive, a possible metric to check for this preciseness could be to divide the number of traces that are in the log and that the mined model can generate by the amount of traces that the mined model can generate. Clearly, a precise mined model could not

[11] Note that in reality we do not know the original (or initial) model. However, the only way to evaluate our results is to assume an initial model. Without an initial model, it is impossible to judge preciseness. In other words, there could be over-fitting or over-generalization, but it would be impossible to judge this.

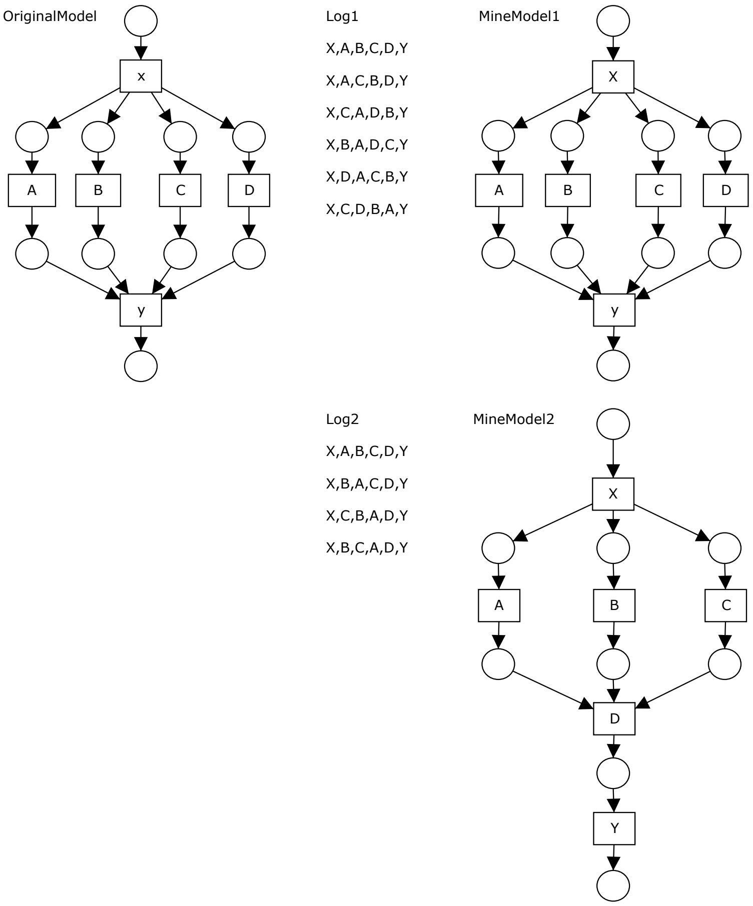


Fig. 9. Example of two mined models that are complete and precise with respect to the logs, but both mined models can generate more traces than the ones in the log. Additionally, the coverability graph of the "MinedModel2" is different from the one of the "OriginalModel".

generate more traces than the ones in the log. Note that this metric would be based on the event log and the mined model. Furthermore, metrics based on the mined and original models would also be possible if the log would be entire. For instance, we could compare the coverability graphs [47] of mapped Petri nets of the mined and the original models. In this case, the mined model would be precise whenever the coverability graphs would be equal. Note that sophisticated notions such as bisimulation [41] and branching bisimulation [22] could also be used. However, none of these metrics are suitable because in real-life applications the log does not hold all possible traces.

For instance, consider the situation illustrated in Figure 9. This figure shows the original model ("OriginalModel"), two synthetic logs ("Log1" and "Log2") and their respective mined models ("MinedModel1" and "MinedModel2"). "Log1" shows that the tasks A, B, C and D are (i) always executed after the task X and before the task Y and (ii) independent of each other. Thus, we can say that the "MinedModel1" is precise with respect to the behavior observed in the “Log1”. However, note that the “MinedModel1”, although precise, can generate more traces than the ones in the “Log1”. A similar reasoning can be done for the “Log2” and the “MinedModel2”. Moreover, the coverability graph of the “MinedModel2” is different from the one of the “OriginalModel”. Actually, based on “Log2”, the “MinedModel2” is more precise than the “OriginalModel”. This illustrates that, when assessing how close the behavior of the mined and original models are, we have to consider the event log that was used by the genetic algorithm. Therefore, we have defined two metrics to quantify how similar the behavior of the original model and the mined model are *based on the event log used during the mining process.*

The two metrics are the *behavioral precision* (B_P) and the *behavioral recall* (B_R). Both metrics are based on the parsing of an event log by the mined model and by the original model. The B_P and B_R metrics are respectively formalized in definitions 13 and 14. These metrics basically work by checking, for the continuous semantics parsing of every task in every process instance of the event log, how many tasks are enabled in the mined model and how many are enabled in the original model. The more enabled tasks the models have in common, the more similar their behaviors are with respect to the event log. The behavioral precision B_P checks how much behavior is allowed by the mined model that is not by the original model. The behavioral recall B_R checks for the opposite. Additionally, both metrics take into account how often a trace occurs in the log. This is especially important when dealing with logs in which some paths are more likely than others, because deviations corresponding the infrequent paths are less important than deviations corresponding to frequent behavior. Note that, assuming a log generated from an original model and a mined model for this log, we can say that the closer their B_P and B_R are to 1, the more similar their behaviors. More specifically, we can say that:

- The mined model is *as precise as* the original model whenever B_P and B_R are equal to 1. This is exactly the situation illustrated in Figure 9 for the “OriginalModel”, the “Log1” and the “MinedModel1”.
- The mined model is *more precise than* the original model whenever $B_P = 1$ and $B_R < 1$. For instance, see the situation illustrated in Figure 9 for the “OriginalModel”, the “Log2” and the “MinedModel2”.
- The mined model is *less precise than* the original model whenever $B_P < 1$ and $B_R = 1$. For instance, see the situation illustrated for the original model in Figure 2, the log in Figure 1, and the mined models in Figure 7.

Definition 13 (Behavioral Precision - B_P) [12] *Let L be an event log. Let CM_o and CM_m be the respective causal matrices for the original (or base)*

[12] For both definitions 13 and 14, whenever the denominator “$|Enabled(CM, \sigma, i)|$” is equal to 0, the whole division is equal to 0. For simplicity reasons, we have omitted this condition from the formulae.

model and for the mined one. Then the behavioral precision $B_P : \mathcal{L} \times \mathcal{CM} \times \mathcal{CM} \rightarrow [0, 1]$ is a function defined as:

$$B_P(L, CM_o, CM_m) =$$

$$\frac{\sum_{\sigma \in L} \left(\frac{L(\sigma)}{|\sigma|} \times \sum_{i=1}^{|\sigma|} \frac{|Enabled(CM_o, \sigma, i) \bigcap Enabled(CM_m, \sigma, i)|}{|Enabled(CM_m, \sigma, i)|} \right)}{\sum_{\sigma \in L} L(\sigma)}$$

where

- *$Enabled(CM, \sigma, i)$ gives the enabled activities at the causal matrix CM just before the parsing of the element at position i in the trace σ. During the parsing a continuous semantics is used (see Section 4.2.1).*

Definition 14 (Behavioral Recall - B_R) *Let L be an event log. Let CM_o and CM_m be the respective causal matrices for the original (or base) model and for the mined one. Then the behavioral recall $B_R : \mathcal{L} \times \mathcal{CM} \times \mathcal{CM} \rightarrow [0, 1]$ is a function defined as:*

$$B_R(L, CM_o, CM_m) =$$

$$\frac{\sum_{\sigma \in L} \left(\frac{L(\sigma)}{|\sigma|} \times \sum_{i=1}^{|\sigma|} \frac{|Enabled(CM_o, \sigma, i) \bigcap Enabled(CM_m, \sigma, i)|}{|Enabled(CM_o, \sigma, i)|} \right)}{\sum_{\sigma \in L} L(\sigma)}$$

Reasoning about the Quality of the Mined Models

When evaluating the quality of a data mining approach (genetic or not), it is common to check if the approach tends to find over-general or over-specific solutions. In our case, the over-general solution is the one that can parse any trace that can be formed from the tasks in a log. This solution has a self-loop for every task in the log. The over-specific solution is the one that has a branch for every *unique* trace in the log. Figure 10 illustrates an over-general and an over-specific solution for the log in Table 1.

The over-general solution *does* belong to the search space considered in this paper. However, this kind of solution can be easily detected by the metrics we have defined so far. Note that, for a given original model CM_o, a log L generated by simulating CM_o, and the mined over-general model CM_m, it always holds that: (i) the over-general model is complete (i.e., $PF_{complete}(L, CM_m) = 1$); (ii) while parsing the traces, all the tasks that are enabled in the original model are also enabled in the over-general model (i.e., $B_R(L, CM_o, CM_m) =$

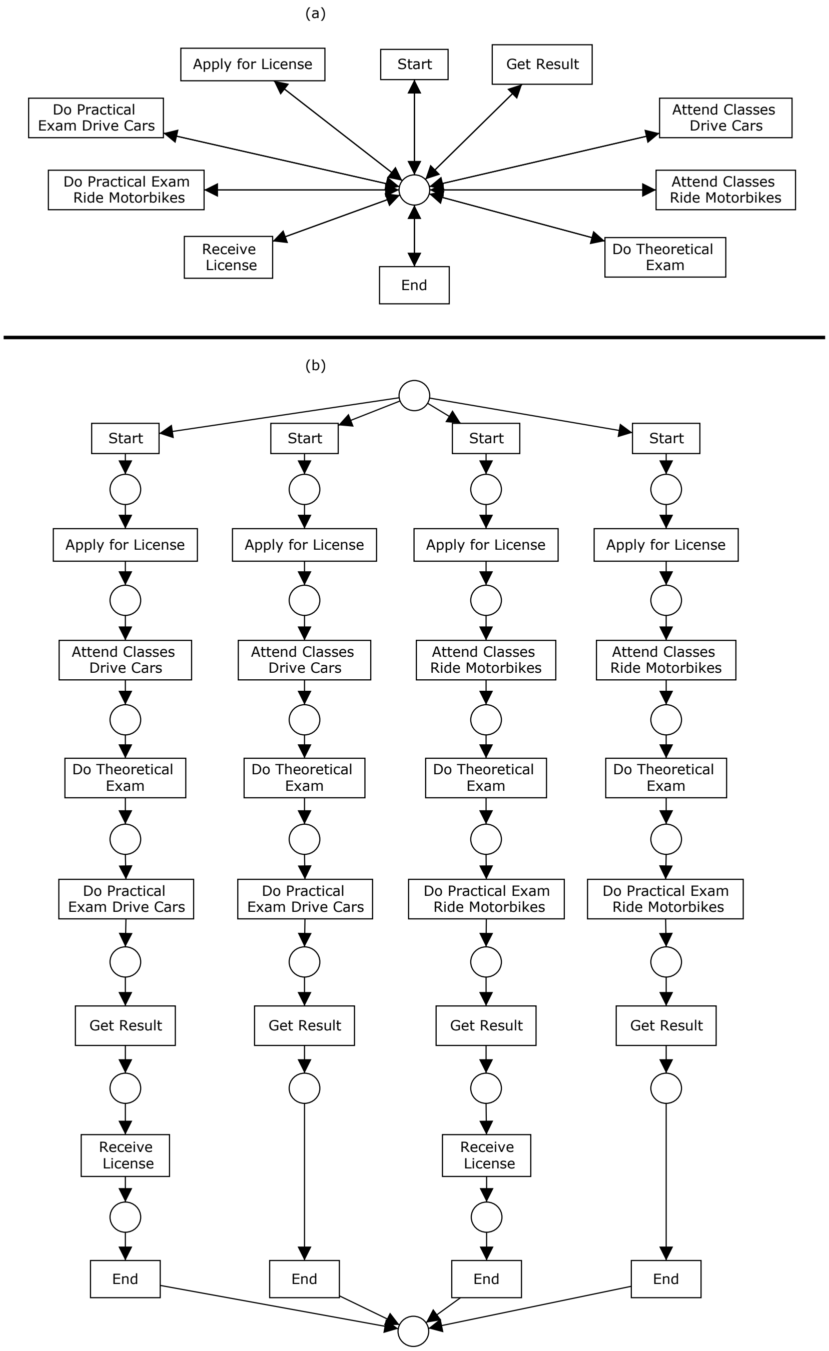

(a)
Apply for License
Start
Get Result
Do Practical Exam Drive Cars
Attend Classes Drive Cars
Do Practical Exam Ride Motorbikes
Attend Classes Ride Motorbikes
Receive License
Do Theoretical Exam
End
(b)
Start
Start
Start
Start
Apply for License
Apply for License
Apply for License
Apply for License
Attend Classes Drive Cars
Attend Classes Drive Cars
Attend Classes Ride Motorbikes
Attend Classes Ride Motorbikes
Do Theoretical Exam
Do Theoretical Exam
Do Theoretical Exam
Do Theoretical Exam
Do Practical Exam Drive Cars
Do Practical Exam Drive Cars
Do Practical Exam Ride Motorbikes
Do Practical Exam Ride Motorbikes
Get Result
Get Result
Get Result
Get Result
Receive License
Receive License
End
End
End
End

Fig. 10. Example of nets that are (a) over-general and (b) over-specific for the log in the Table 1.

1); and (iii) while parsing the traces, all the tasks of the over-general model are always enabled, i.e., the formula of the behavioral precision (see Definition 13) can be simplified to the formula in Equation 1. These three remarks are used to detect over-general mined models during the experiments analysis.

$$B_P(L, CM_o, CM_m) = \frac{\sum\limits_{\sigma \in L} \left(\frac{L(\sigma)}{|\sigma|} \times \sum\limits_{i=1}^{|\sigma|} \frac{|Enabled(CM_o, \sigma, i)|}{|A_m|} \right)}{\sum\limits_{\sigma \in L} L(\sigma)} \tag{1}$$

Contrary to the over-general solution, the over-specific one *does not* belong to our search space because our internal representation (the causal matrix) does not support duplicate tasks. However, because our fitness only looks for the complete and precise behavior (not the minimal representation, like the works on *Minimal Description Length (MDL)* [28]), it is still important to check how similar the structures of the mined model and the original one are. Differences in the structure may point out another good solution or an overly complex solution. For instance, have a look at the model in Figure 11. This net is complete and precise from a behavioral point of view, but it contains extra unnecessary places. Note that the places "p12" and "p13" could be removed from the net without changing its behavior. In other words, "p12" and "p13" are implicit places [5]. Actually, because the places do not affect the net behavior, all the nets in figures 2, 8 and 11 have the same fitness. However, a metric that checks the structure of a net would, for instance, point out that the net in Figure 11 is a "superstructure" of the net in Figure 2, and has many elements in common with the net in Figure 8. So, even when we know that the over-specific solution is out of the search space defined in this paper, it is interesting to get a feeling about the structure of the mined models. That is why we developed two metrics to assess how much the mined and original model have in common *from a structural point of view.*

The two metrics are the *structural precision* (S_P) and the *structural recall* (S_R). Both metrics are based on the *causality relations* of the mined and original models, and were adapted from the precision and recall metrics presented in [49]. The S_P and S_R metrics are respectively formalized in definitions 13 and 16. These metrics basically work by checking how many causality relations the mined and the original models have in common. The more causality relations the two models have in common, the more similar their structures are. The structural precision assess how many causality relations the mined model has that are not in the original model. The structural recall works the other way around. Note that the structural similarity performed by these metrics does not consider the semantics of the split/join points. We have done so because the causality relations are the core of our genetic material (see Subsection 4.4). The semantics of the split/join tasks can only be correctly captured if the right dependencies (or causality relations) between the tasks in the log are also in place.

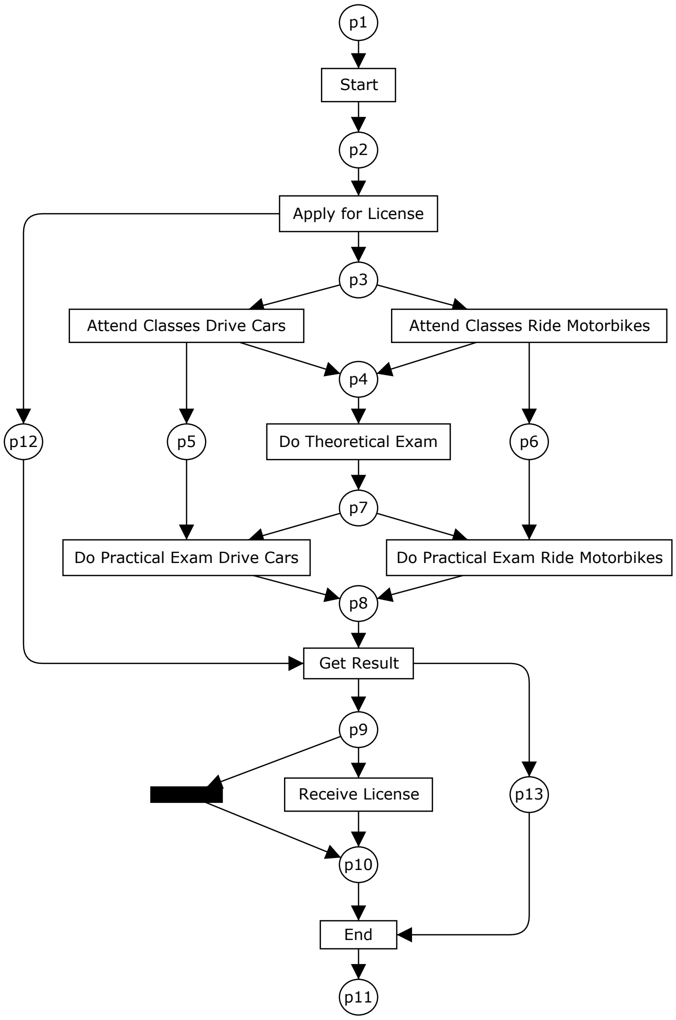

p1
Start
p2
Apply for License
p3
Attend Classes Drive Cars
Attend Classes Ride Motorbikes
p4
p12
p5
Do Theoretical Exam
p6
p7
Do Practical Exam Drive Cars
Do Practical Exam Ride Motorbikes
p8
Get Result
p9
Receive License
p13
p10
End
p11

Fig. 11. Example of a net that that is behavioral precise and complete w.r.t. the log in the Table 1, but that contains extra unnecessary (implicit) places (p12 and p13).

Definition 15 (Structural Precision - S_P**)** [13] *Let* CM_o *and* CM_m *be the respective causal matrices for the original and the mined models. The structural precision* $S_P : \mathcal{CM} \times \mathcal{CM} \rightarrow [0,1]$ *is a function defined as:*

$$S_P(CM_o, CM_m) = \frac{|C_o \cap C_m|}{|C_m|}$$

Definition 16 (Structural Recall - S_R**)** *Let* CM_o *and* CM_m *be the respective causal matrices for the original and the mined model. The structural recall*

[13] For both definitions 13 and 16, whenever the denominator "$|C|$" is equal to 0, the whole division is equal to 0. For simplicity reasons, we have omitted this condition from the formulae.

$S_R : \mathcal{CM} \times \mathcal{CM} \rightarrow [0, 1]$ *is a function defined as:*

$$S_R(CM_o, CM_m) = \frac{|C_o \cap C_m|}{|C_o|}$$

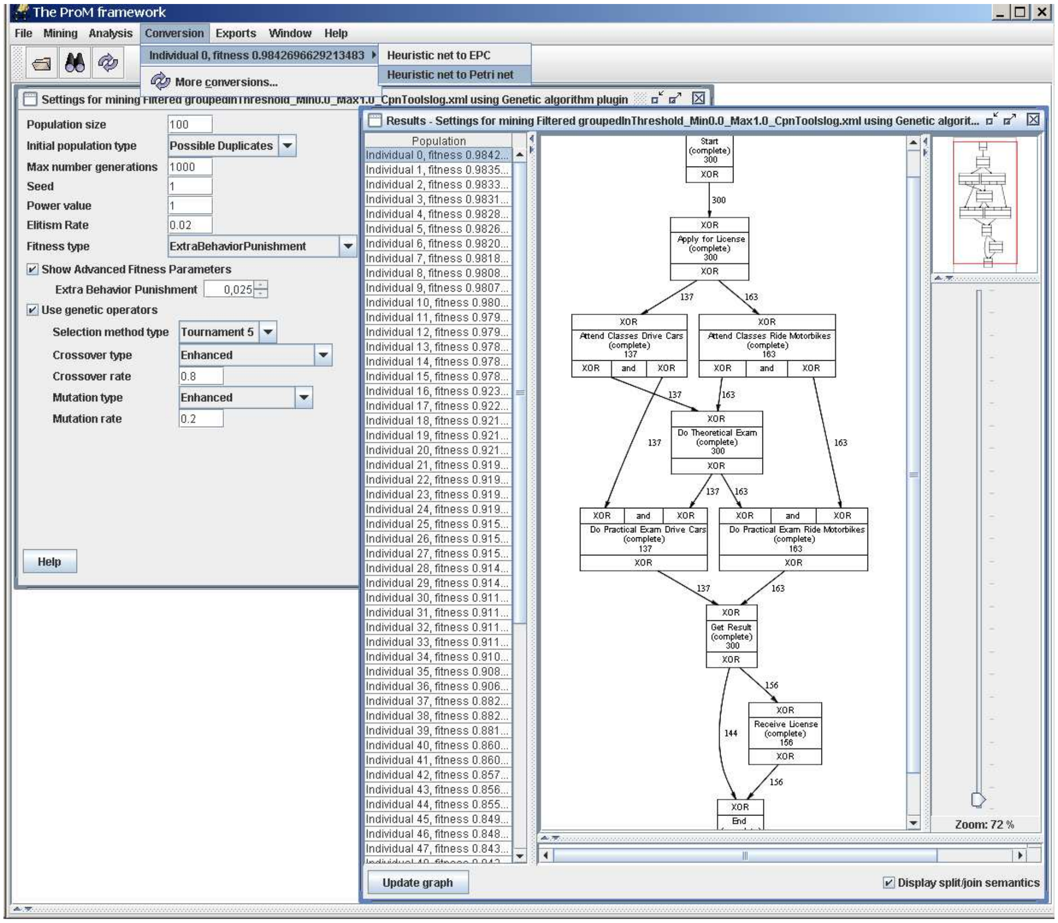
When the original and mined models have behavioral metrics B_R and B_P that are equal to 1, the S_R and S_P show how similar the structure of these models are. For instance, for the original model in Figure 2, the structural metrics would indicate that a mined model like the one in Figure 8 differs from the original one by the same amount of causality relations ($S_R = S_P$), and a mined model like the one in Figure 11 has extra causality relations($S_R = 1$ and $S_P < 1$).

Recapitulation of the Analysis Metrics

This section has presented the five metrics that are used to analyse the experiments in this paper: the partial fitness for the completeness requirement ($PF_{complete}$), the behavioral precision (B_P), the behavioral recall (B_R), the structural precision (S_P) and the structural recall (S_R). The $PF_{complete}$ quantifies how complete a mined model is. The B_P and B_R measure how precise the mined model is. The S_P and S_R express if the mined model has an overly complex structure or not. These metrics are complementary and should be considered together during the experiments analysis. For instance, for our experiments, the mined model is as complete and precise as the original model whenever the metrics $PF_{complete}$, B_P and B_R are equal to 1. More specifically, the mined model is exactly like the original model when all the five metrics are equal to 1. As a general rule, the closer the values of the five metrics are to 1, the better.

6 Experiments and Results

This section explains how we conducted the experiments and, most important of all, how we analyzed the quality of the models that the genetic algorithm mined. To conduct the experiments we needed (i) to implement our genetic algorithm and (ii) a set of event logs. The *genetic algorithm* described in this paper *is implemented as the "Genetic algorithm plug-in"* in the ProM framework (see Figure 12). The ProM framework is available at www.processmining.org and supports the development of plug-ins to mine event logs. Although in this paper we focus on the "Genetic algorithm plug-in", the ProM framework offers other plug-ins like, for instance, the "Social network miner plug-in" [2] and "Conformance checker plug-in" [53]. The *logs* used in our experiments are *synthetic*. In brief, we built the model (or copied it from some related work)



The ProM framework
File
Mining
Analysis
Conversion
Exports
Window
Help
Individual 0, fitness 0.9842696629213483
Heuristic net to EPC
Heuristic net to Petri net
More conversions...
Help
Update graph
Start (complete) 300
XOR
300
Apply for License (complete) 300
137
163
Attend Classes Drive Cars (complete) 137
and
Attend Classes Ride Motorbikes (complete) 163
Do Theoretical Exam (complete) 300
Do Practical Exam Drive Cars (complete) 137
Do Practical Exam Ride Motorbikes (complete) 163
Get Result (complete) 300
156
144
Receive License (complete) 156
End
Zoom: 72 %

Settings for mining Filtered groupedInThreshold_Min0.0_Max1.0_CpnToolslog.xml using Genetic algorithm plugin

Population size 100
Initial population type Possible Duplicates
Max number generations 1000
Seed 1
Power value 1
Elitism Rate 0.02
Fitness type ExtraBehaviorPunishment
☑ Show Advanced Fitness Parameters
Extra Behavior Punishment 0,025
☑ Use genetic operators
Selection method type Tournament 5
Crossover type Enhanced
Crossover rate 0.8
Mutation type Enhanced
Mutation rate 0.2

Results - Settings for mining Filtered groupedInThreshold_Min0.0_Max1.0_CpnToolslog.xml using Genetic algorit...

Population
Individual 0, fitness 0.9842...
Individual 1, fitness 0.9835...
Individual 2, fitness 0.9833...
Individual 3, fitness 0.9831...
Individual 4, fitness 0.9828...
Individual 5, fitness 0.9826...
Individual 6, fitness 0.9820...
Individual 7, fitness 0.9818...
Individual 8, fitness 0.9808...
Individual 9, fitness 0.9807...
Individual 10, fitness 0.980...
Individual 11, fitness 0.979...
Individual 12, fitness 0.979...
Individual 13, fitness 0.978...
Individual 14, fitness 0.978...
Individual 15, fitness 0.978...
Individual 16, fitness 0.923...
Individual 17, fitness 0.922...
Individual 18, fitness 0.921...
Individual 19, fitness 0.921...
Individual 20, fitness 0.921...
Individual 21, fitness 0.919...
Individual 22, fitness 0.919...
Individual 23, fitness 0.919...
Individual 24, fitness 0.919...
Individual 25, fitness 0.915...
Individual 26, fitness 0.915...
Individual 27, fitness 0.915...
Individual 28, fitness 0.914...
Individual 29, fitness 0.914...
Individual 30, fitness 0.911...
Individual 31, fitness 0.911...
Individual 32, fitness 0.911...
Individual 33, fitness 0.911...
Individual 34, fitness 0.910...
Individual 35, fitness 0.908...
Individual 36, fitness 0.906...
Individual 37, fitness 0.882...
Individual 38, fitness 0.882...
Individual 39, fitness 0.881...
Individual 40, fitness 0.860...
Individual 41, fitness 0.860...
Individual 42, fitness 0.857...
Individual 43, fitness 0.856...
Individual 44, fitness 0.855...
Individual 45, fitness 0.849...
Individual 46, fitness 0.848...
Individual 47, fitness 0.843...

☑ Display split/join semantics

Fig. 12. Screenshot of the "Genetic algorithm plug-in" in the ProM framework. This screenshot shows the result of mining an event log like the one in Table 1. This log has 300 process instances in total. The left-side window shows the configuration parameters (see Section 6.1). The right-side window shows the best mined individual (or causal matrix). Additionally, in the menu bar we show how to convert this individual (called "Heuristics Net" in the ProM framework) to a Petri net.

and simulated it to create a synthetic event log. We then run the genetic algorithm over these sets of logs. Once the genetic algorithm finished the mining process, we analyzed the results.

When conducting the experiments we were interested in getting an indication of two main points: (i) how the heuristics and genetic operators influence the quality of mined models, and (ii) if the fitness measure indeed guides the search towards individuals that reflect the most frequent behavior in the log. The first point was investigated by executing the genetic algorithm over synthetic *noise-free* log. Details about these experiments are explained in Subsection 6.1. The second point was investigated by running the genetic algorithm over synthetic *noisy* logs. Details about the experiments with noisy logs are reported in Subsection 6.2.

Setup

The genetic algorithm was tested over noise-free event logs from 25 different process models. These models contain constructs like sequence, choice, parallelism, loops, non-free-choice and invisible tasks. From the 25 models, 6 were copied from the models in [30]. The other models were created by the authors. The models had between 6 and 42 tasks[14] . Every event log was randomly generated and contained 300 process instances. To speed up the computational time of the genetic algorithm, the similar traces were grouped into a single one and a counter was associated to inform how often the trace occurs. The similarity criterion was the local information in the trace. Traces with the same direct left and right neighbors for every element were grouped together. Besides, *to test how strong the use of the genetic operators and the heuristics influence the results*, we set up four scenarios while running the genetic algorithm: ("Scenario I") without heuristics to build the initial population and without genetic operators [15]; ("Scenario II") with heuristics, but without the genetic operators; ("Scenario III") without heuristics, but with genetic operators; and ("Scenario IV") with heuristics and genetic operators. For every log, 50 runs were executed for every scenario. Every run had a population size of 100 individuals, at most 1000 generations, an elite of 2 individuals and a κ of 0.025 (see Definition 12). The experiments with heuristics used a power value of 1 while building the initial population (see Subsection 4.1.1). The experiments with the genetic operators have a respective crossover and mutation probabilities of 0.8 and 0.2 (see the respective subsections 4.4.1 and 4.4.2). All the experiments were run using the ProM framework, our tool set that can be obtained via www.processmining.org. We implemented the genetic algorithm and the metrics (cf. Section 5) described in this paper as plug-ins for this framework.

Results

The results are reported in Figures[16] 13 to 23. Figures 13 to 19 show the average values of the analysis metrics (cf. Section 5) for the mined models. Figures 20 and 21 indicate how the fitness of the best mined model evolved

[14] Table A.1 (cf. Appendix A) provides more details about the characteristics of these models.

[15] This scenario is a random generation of individuals. The aim of experimenting with this scenario is to assess if the use of genetic operators and/or heuristics is better than the pure random generation of individuals, given the same limited amount of computational time.

[16] The unmarked points in these figures correspond to experiments that were interrupted because they were taking more than 6 hours to process one seed.

over generations. Figures 22 and 23 show how much computational time was required, on average, per run of the genetic algorithm[17]. Overall, the results indicate that the scenario for the hybrid genetic algorithm (Scenario IV) is superior to the other scenarios in all aspects. More specifically, the results show that:

- Any approach (scenarios II to IV) is better than the pure random generation of individuals (Scenario I) (cf. Figures 13 to 19).
- Scenario II and IV mined more complete and precise models than the other scenarios (cf. Figure 19).
- The hybrid genetic algorithm (Scenario IV) works the best. This approach combines the strong ability of the heuristics to correctly capture the *local* causality relations with the benefits of using the genetic operators (especially mutation) to introduce the non-local causality relations. For instance, consider the results for the nets `a6nfc`, `driversLicense` and `herbstFig6p36`. All these nets have non-local non-free-choice constructs. Note that, for these three nets, the results for Scenario II (cf. Figure 14) have a much lower behavioral precision than for Scenario IV (cf. Figure 16). This illustrates the importance of the genetic operators to insert the non-local causality relations.
- In general, the hybrid genetic algorithm (Scenario IV) mines more complete models *that are also precise* than the other scenarios (see Figure 19). In fact, except for net `a7`, Scenario IV is the only configuration that mined a complete and precise model for at least one of the runs (cf. Figure 19). This shows that this scenario finds mined models that are complete and precise faster than the other scenarios.
- Nets with short parallel branches (like `parallel5`, `a7` and `a5`) are more difficult to mine. This is due to the probabilistic nature of the genetic algorithm. Recall that the fitness measure always benefits the individuals that can parse the most frequent behavior in the log. So, in parallel situations, it is often the case that the algorithm goes for individuals that show the most frequent interleaving patterns in the log.
- Although Scenario II led to better results than Scenario III, it is not fair to compare them. The reason is that Scenario III *starts from scratch*, in the sense that its initial population is randomly built, while Scenario II *is strongly helped by good heuristics* to detect local dependencies. In our experiments, Scenario III is used to show that (i) the use of the genetic operators improves the results (and this is indeed the case, since Scenario III gave better results than Scenario I), and (ii) the genetic operators help in mining non-local dependencies (again, note that the results for the nets with non-local non-free-choice constructs - `a6nfc` and `driversLicense` - are better in Scenario III than in Scenario II). Thus, in short, Scenario III

[17] The experiments were run in a Intel® Pentium® 4 CPU 3.40 GHz 3.39 GHz, 1.99 GB RAM, with Microsoft Windows XP 2002.

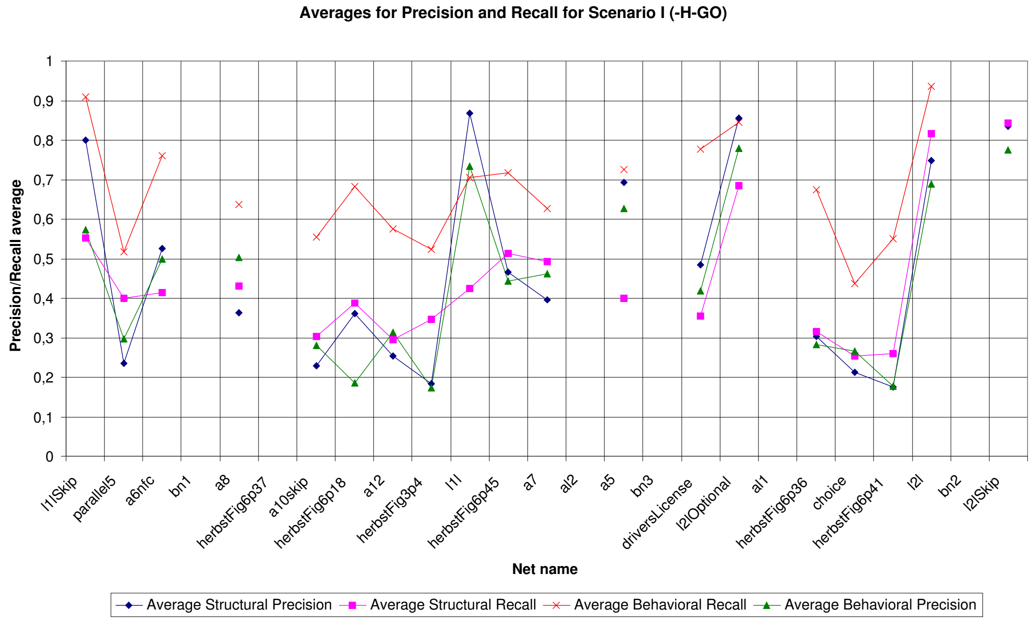


Fig. 13. Average precision and recall values of the results for **Scenario I** (without heuristics to build the initial population and without using genetic operators to build the following populations).

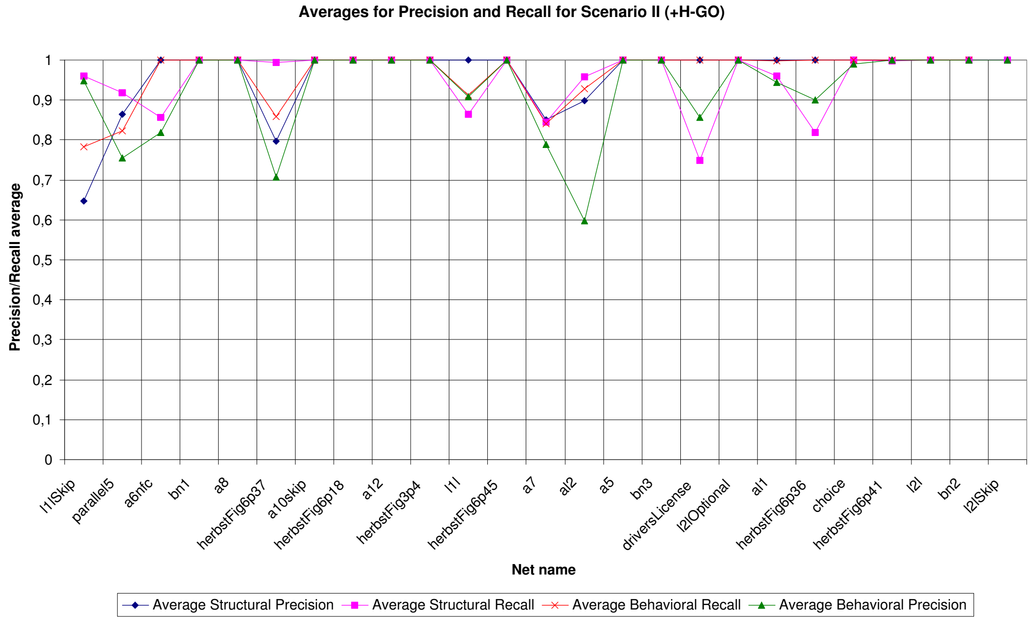


Fig. 14. Average precision and recall values of the results for **Scenario II** (with heuristics to build the initial population, but without using genetic operators to build the following populations).

shows that the GA was going on the right track, but it would need more iterations to reach or outperform the results of Scenario II for all nets.

- The use of genetic operators makes the population converge to better models

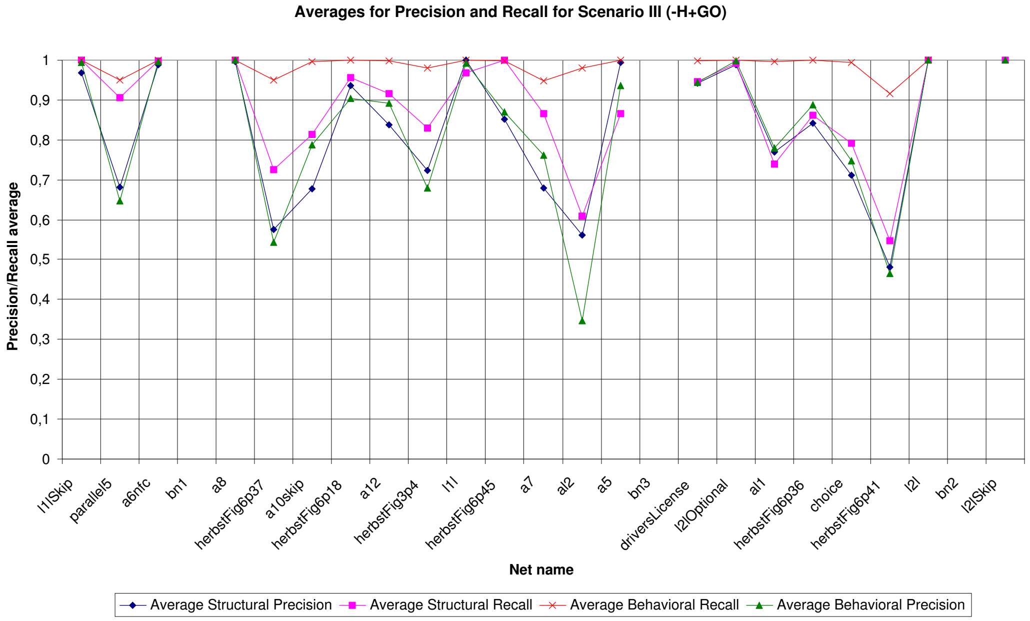


Fig. 15. Average precision and recall values of the results for **Scenario III** (without heuristics to build the initial population, but using genetic operators to build the following populations).

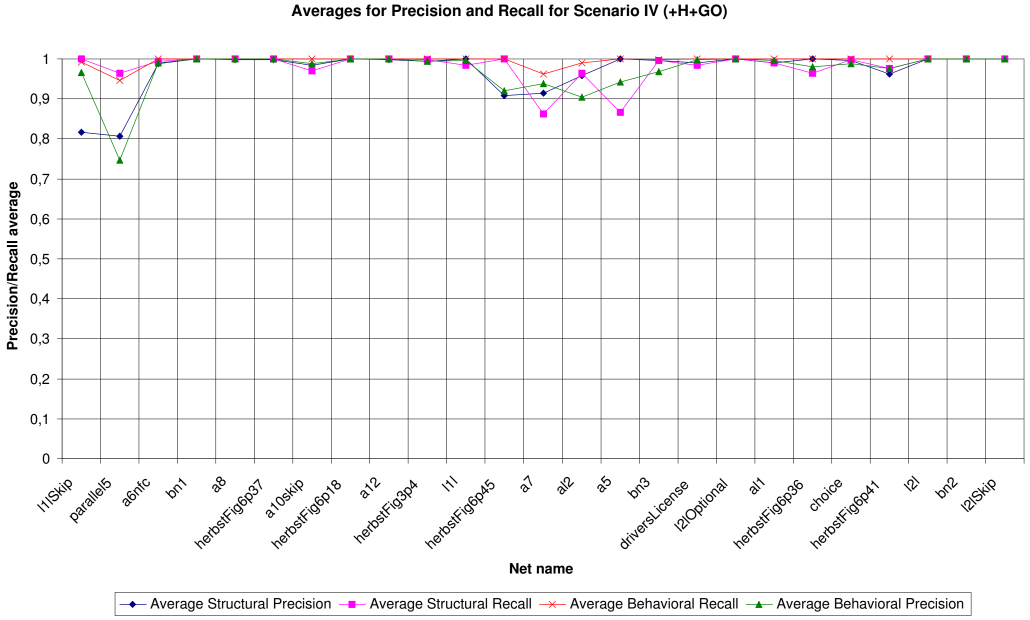


Fig. 16. Average precision and recall values of the results for **Scenario IV** (with heuristics to build the initial population and using genetic operators to build the following populations).

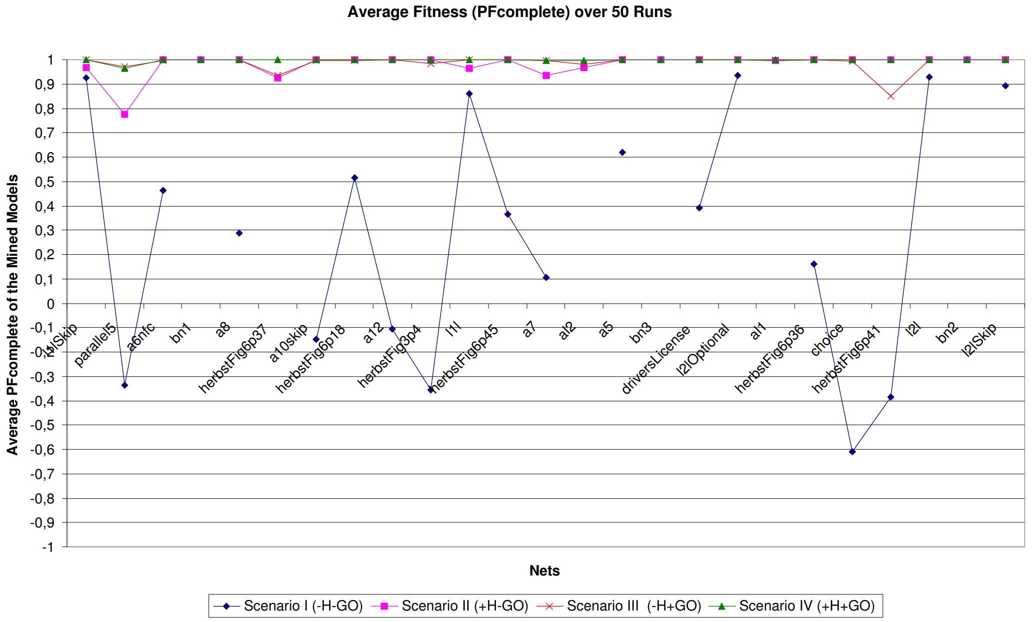


Fig. 17. Average fitness ($PF_{complete}$) values of the mined models for 50 runs.

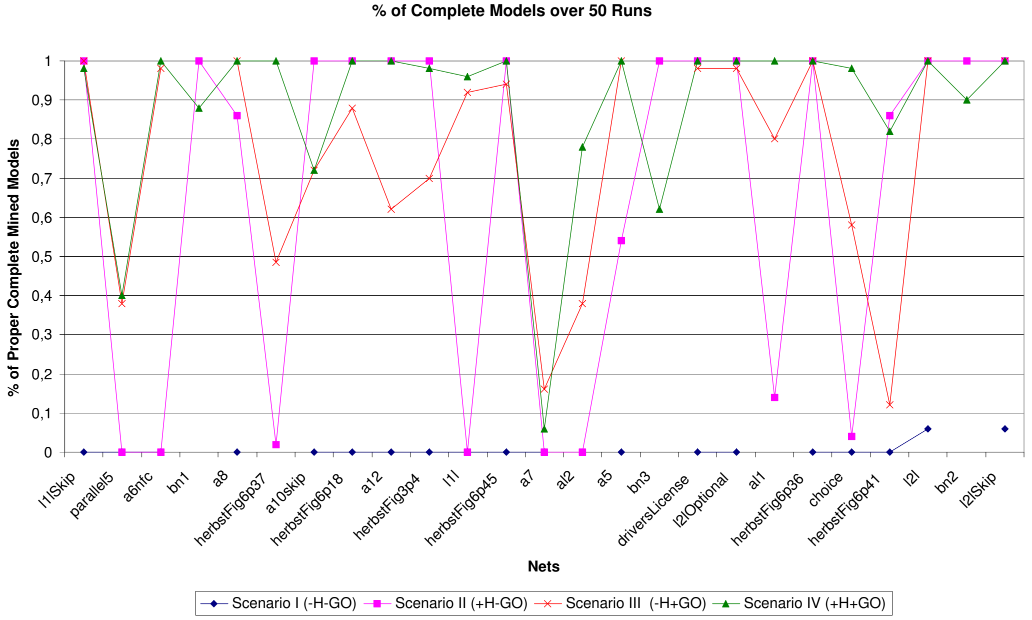


Fig. 18. Percentage of the mined models that proper complete ($PF_{complete} = 1$) over 50 runs.

in a faster pace, as illustrated in figures 20 and 21. Note that for Scenario III and IV, in many situations the best individuals have a fitness superior to 0.90 already at generation 100. Furthermore, the results in these figures show that the heuristics used to build the initial population indeed capture many of the correct causality relations. Note that the best fitness for individuals at generation 0 is already bigger than 0.0 for all nets in scenarios II and IV,

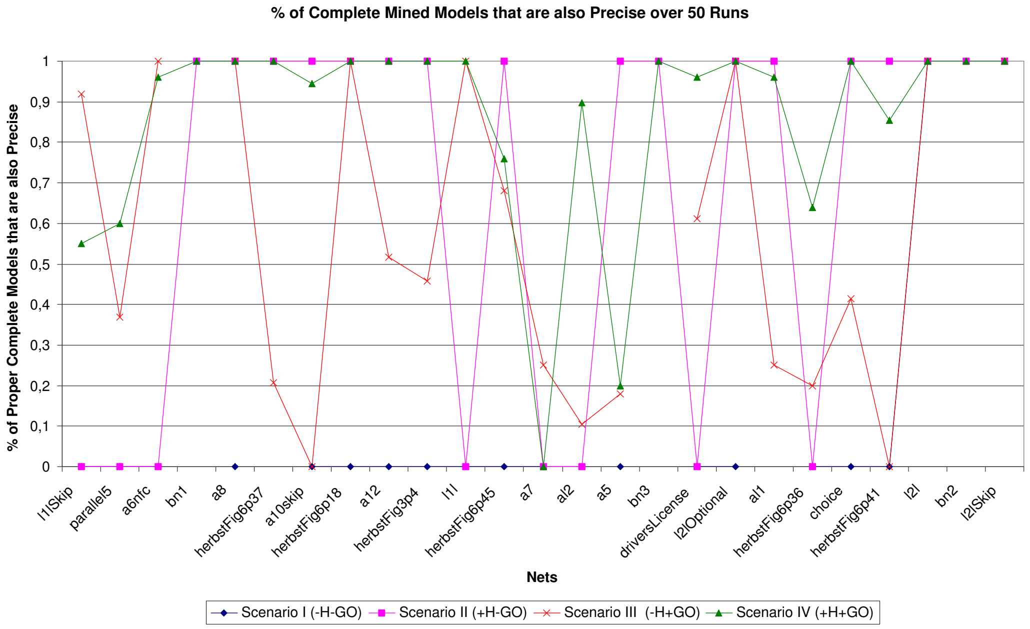


Fig. 19. Percentage of the complete mined models ($PF_{complete} = 1$) that are also precise ($B_P = 1$) over 50 runs.

and this is not the case for scenarios I and III.

- Many of the runs took on average less than 15 minutes (cf. figures 22 and 23). However, logs from nets with loops and/or parallel constructs tend to take more time. These nets usually allow for more interleaving situations and, therefore, have bigger logs. Since the fitness is calculated by replaying the logs in the individuals of a population, the genetic algorithm takes longer for these models. As a general remark, the more different traces and tasks a log contains, the higher will be the computational time of the genetic algorithm.

The next subsection shows the results for the experiments with noisy logs.

6.2 *Experiments with Noisy Logs*

Before reporting on the experiments with noisy logs, we explain the approach we chose to handle mined models from these logs. Noise can be defined as *low-frequent incorrect* behavior in the log. A log may contain noise because some of its traces are incomplete (e.g., they correspond to running cases in the system that have not been completed yet), or the traces reflect incorrect behavior (e.g., due to some temporal system misconfiguration). Either way, the presence of noise may hinder the correct discovery of a process model. Noisy behavior is typically difficult to detect because it cannot be easily distinguished from other *low-frequent correct* behavior in the log (for instance, the execution

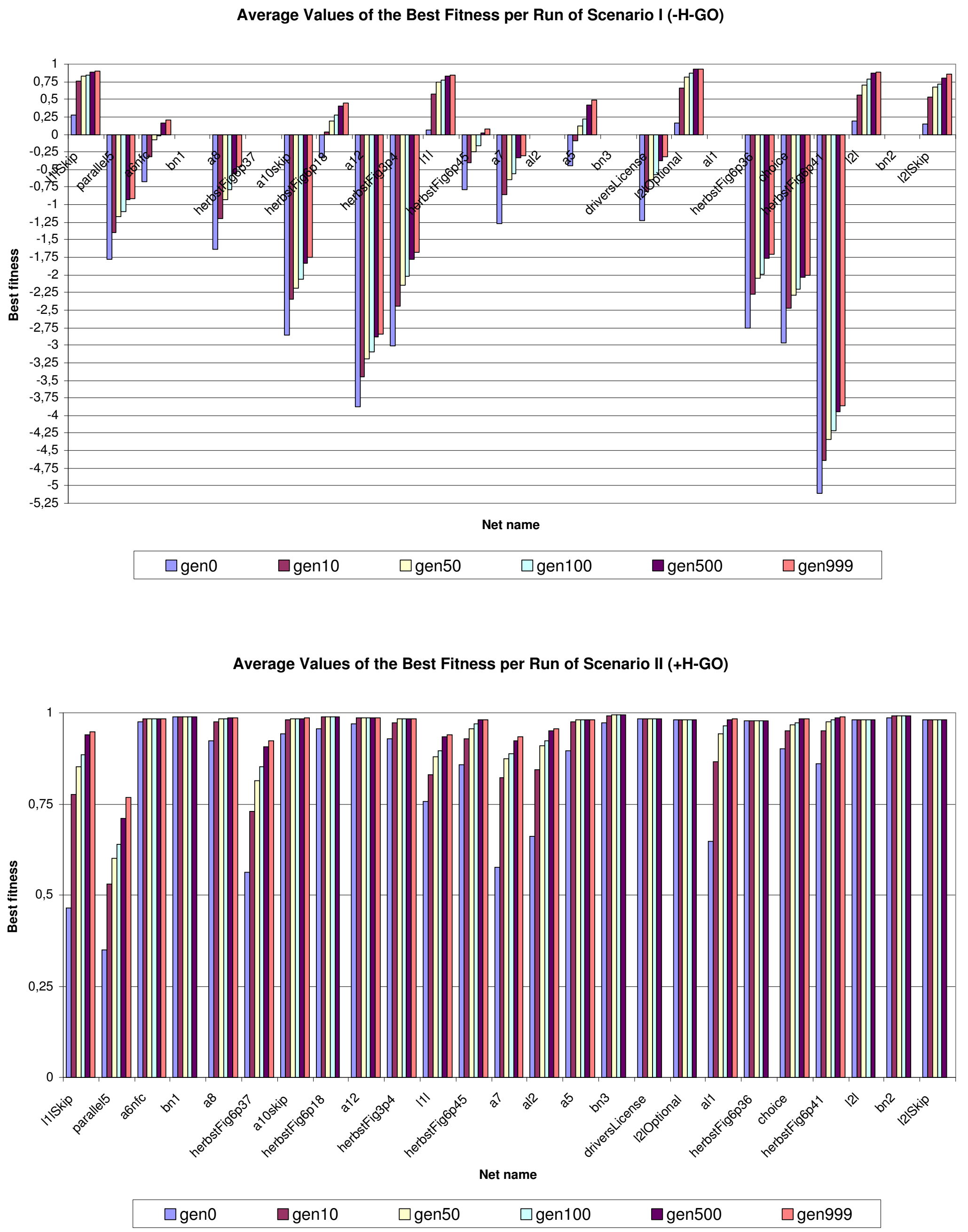
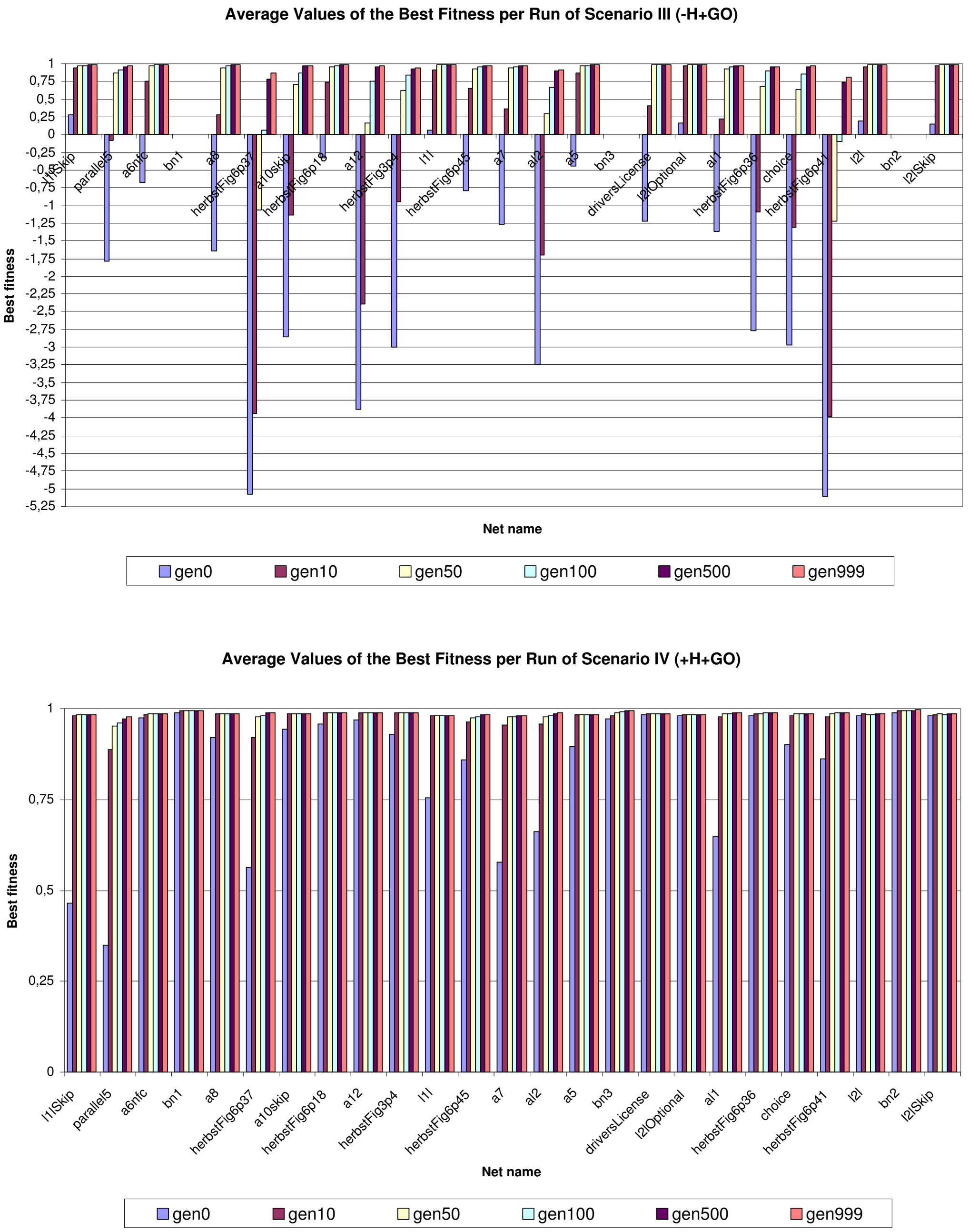


Fig. 20. Average values of the best fitness per run of scenarios I (top graph) and II (bottom graph).

of exceptional paths in the process). For this reason, and because the genetic algorithm is designed to always benefit individuals that can correctly parse the *most frequent* behavior in the log, we have opted for a post-processing step to "clean" mined models from the effects of noise. In short, this post-processing step works by *pruning the arcs of a (mined) model that are used fewer times than a certain threshold*.


Average Values of the Best Fitness per Run of Scenario III (-H+GO)

Average Values of the Best Fitness per Run of Scenario IV (+H+GO)

Fig. 21. Average values of the best fitness per run of scenarios III (top graph) and IV (bottom graph).

The main advantage of a post-pruning step is that, because it works independently of the process mining algorithm, it does not avoid the discovery of low-frequent behavior. Thus, if the mined low-frequent behavior is a correct one, it can remain in the model. If the mined low-frequent behavior corresponds to noisy behavior, the end user has the possibility to clean the mined model. Furthermore, arc post-pruning can also be used over any model to get a more concise view (in terms of the number of arcs in the model) of the

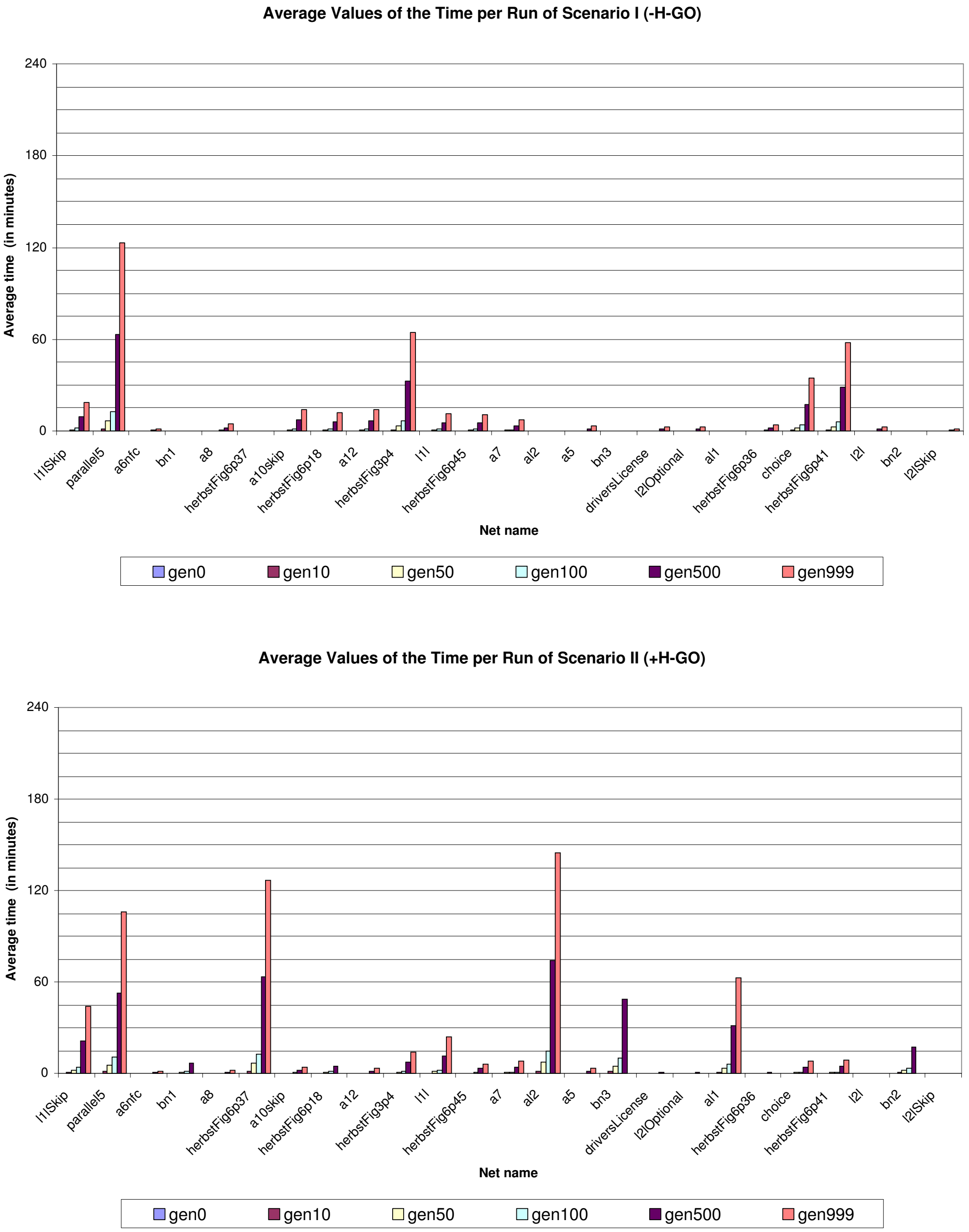


Fig. 22. Average values of the time per run of scenarios I (top graph) and II (bottom graph).

most frequent behavior. As a final remark, we point out that arc post-pruning is also the approach adopted by the other related process mining techniques in [6,10,25,32] to clean mined models. The remainder of this section provides more details about this arc post-pruning approach, the experiments setup and results. The aim of the experiments is to get an indication of how sensitive te genetic algorithm is to different noise types.

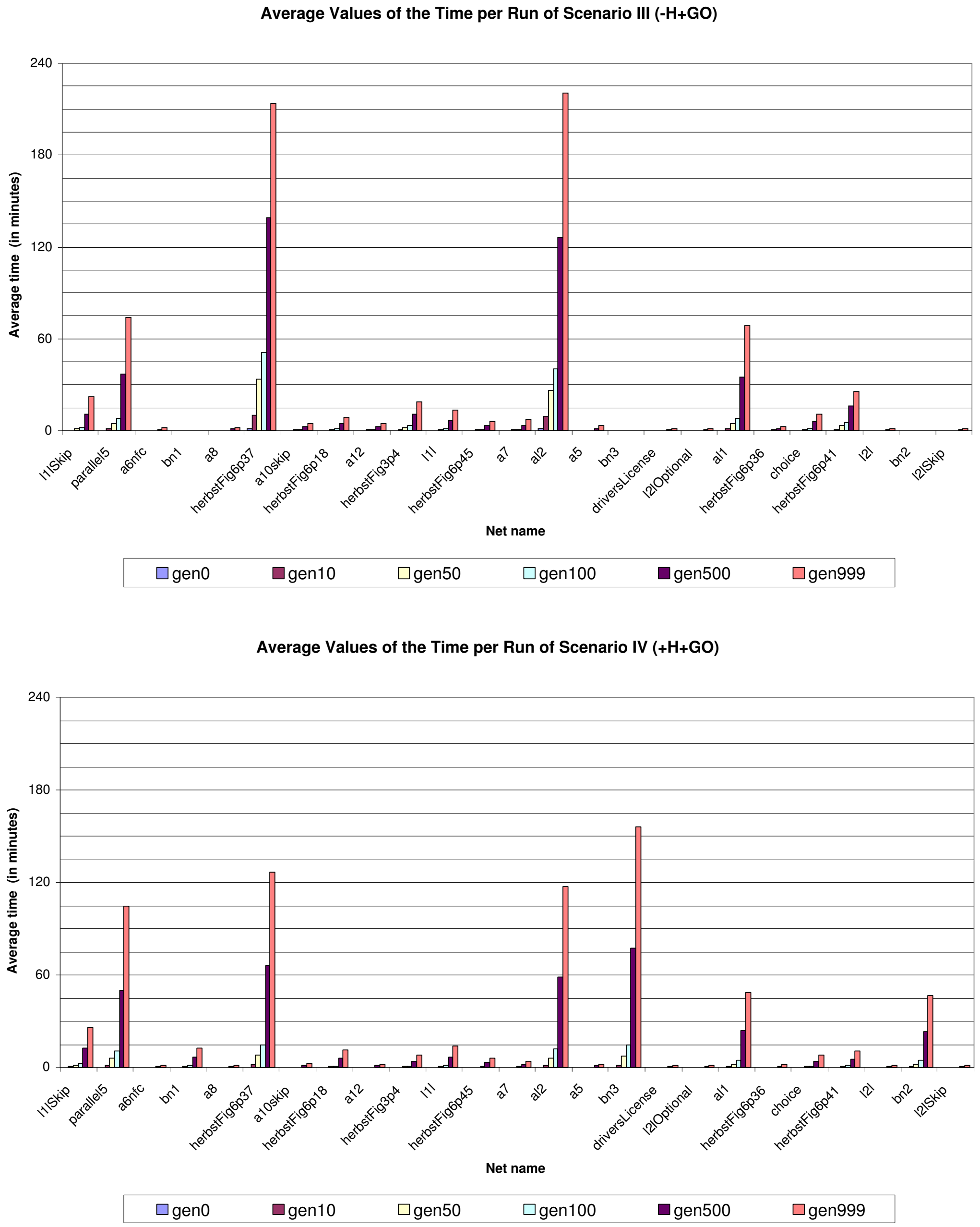


Fig. 23. Average values of the time per run of scenarios III (top graph) and IV (bottom graph).

Post Pruning

The post-pruning step removes the arcs that are used fewer times than a given threshold from a (mined) model. The threshold refers to the arc usage percentage. The arc usage indicates the number of times that an arc (or dependency) is used when a log is replayed by an individual. The arc usage percentage defined by the threshold is relative to the most frequently used arc. As an illustration, assume that the most frequent arc usage of a mined model to a given log is 300. If the threshold is set for 5%, all arcs of this model that are used 15 or fewer times are removed during the post-pruning step. This situation is depicted in Figure 24. The mined model is in Figure 24(a). The pruned model is in Figure 24(b). Notice that the arcs of the mined model that were used (from left to right) 7, 5, 3 and 6 times are not shown in the pruned model. When the removal of arcs leads to dangling activities (i.e., activities without ingoing and outgoing arcs), these activities are also omitted in the post-pruned model.

Setup

The genetic algorithm was tested over noisy logs from 5 different process models: a12, bn1, herbstFig3p4, herbstFig6p36 and herbstFig6p37. These models[18] contain constructs like sequences, choices, parallelism, structured loops and non-local non-free-choice constructs, and have between 12 and 42 tasks. The noise-free log of every net has 300 traces (actually, these are the same noise-free logs used during the experiments reported in Section 6.1). For every noise-free log, 12 noisy logs were generated: 6 logs with 5% noise and 6 logs with 10% noise. The 6 noise types used are: *missing head*, *missing body*, *missing tail*, *swap tasks*, *remove task* and *mix all*. These noise types are the ones described in [36]. If we assume a trace $\sigma = t_1 ... t_{n-1} t_n$, these noise types behave as follows. *Missing head, body* and *tail* respectively randomly remove sub-traces of tasks in the head, body and tail of σ. The head goes from t_1 to $t_{n/3}$ [19]. The body goes from $t_{(n/3)+1}$ to $t_{(2n/3)}$. The tail goes from $t_{(2n/3)+1}$ to t_n. The removed sub-traces contain at least one task and at most all the tasks in the head, body or tail. *Swap task* exchanges two tasks in σ. *Remove task* randomly removes *one* task from σ. *Mix all* randomly performs (with the same probability) one of the other 5 noise types to a traces in a log. Real life logs will typically contain mixed noise. However, the separation between the noise types allows us to better assess how the different noise types affect the genetic algorithm.

To speed up the computational time of the genetic algorithm, the similar traces were grouped into a single one and a weight was added to indicate how often the trace occurs. Traces with the same sequence of tasks were grouped together. For every noisy log, 50 runs were executed. The configuration of every run is the same used for the noise-free experiments of the Scenario IV (see Section 6.1). After a run was complete, the mined model was used as input

[18] The main motivation to select only five models is that, as shown in Section 6.1, a run of the genetic algorithm is time consuming and, as we explain further in the text, 12 noisy logs were generated per net. Thus, the choice for five models is a pragmatic one.

[19] The division $n/3$ is rounded to the largest double value that is not greater than $n/3$ and is equal to a mathematical integer.

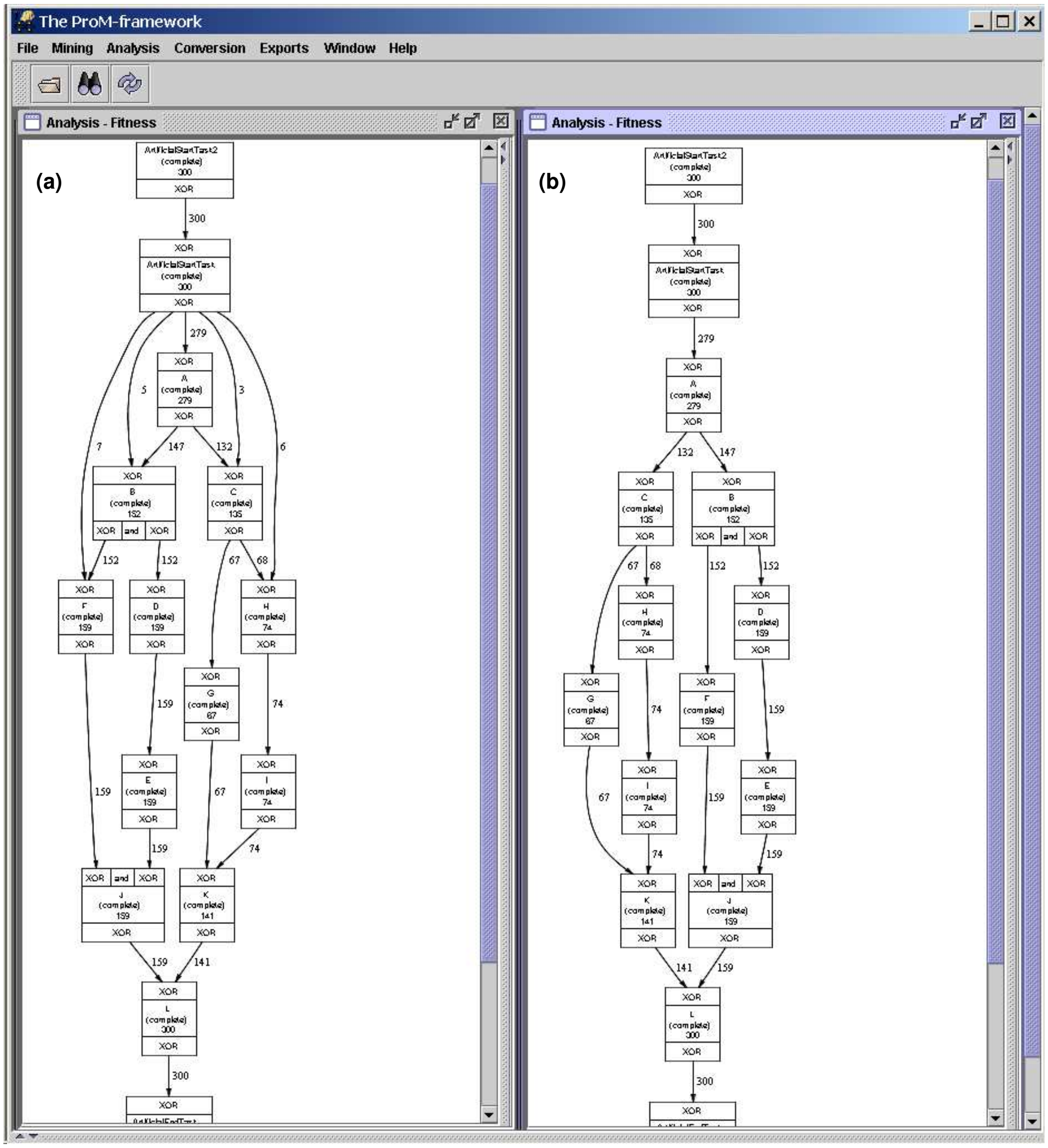


Fig. 24. Illustration of applying post-pruning to arcs of a mined model. The mined model is in (a), and the resulting post-pruned model is in (b). The numbers next to the arcs in these models inform how often these arcs have been used while replaying the log for the models. The post-pruning illustrated here has a threshold of 5%. Thus, since the highest arc usage of the mined model in (a) is 300, all of its arcs that are used 15 or fewer times are not shown in the resulting pruned model in (b).

for a post-processing step to prune its arcs. Every mined model went two post-pruning steps: one to prune with a threshold of 5% and another to prune with a threshold of 10%. So, when analyzing the results, we look at (i) the mined model returned by the genetic algorithm, (ii) the model after applying 5% pruning, and (iii) the model after 10% pruning. The post-processing step is implemented as the *Prune Arcs* analysis plug-in in the ProM framework.

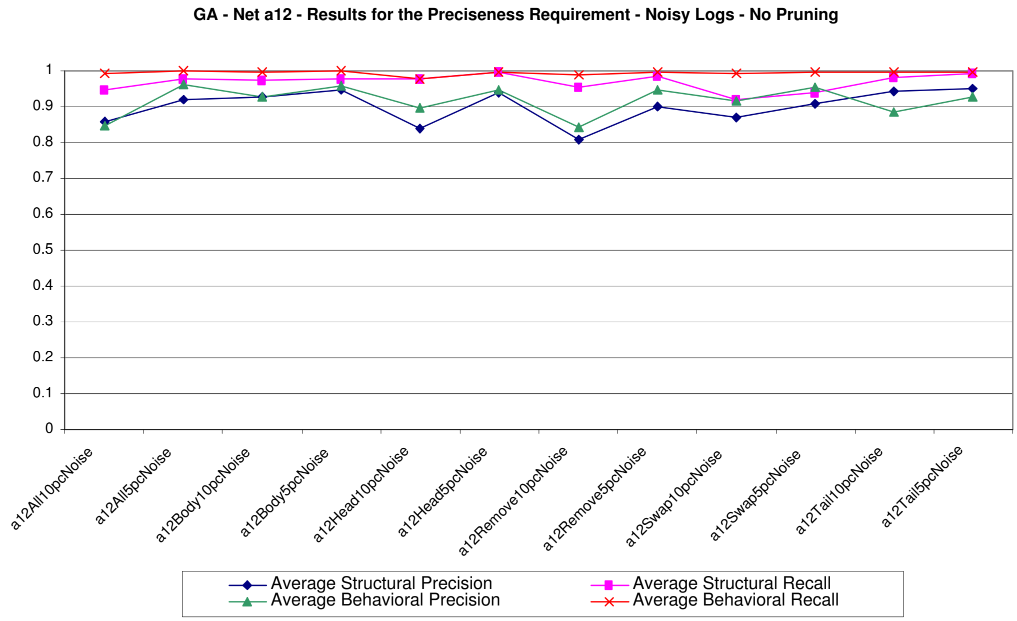
Results

The results for the experiments with logs of the net a12 are in Figure 25 to 28. We only show the results for the net a12 because the obtained results for the other nets lead to the same conclusions that can be drawn based on the analysis of the results for a12. Every figure plots the results before and after pruning. However, we have omitted the results for 10% arc-pruning because the results are just like the results for 5% arc-pruning. Furthermore, for every graph, the x-axis shows, for a given net (or original model), the noise type and the percentage of noise in the log. For instance, a12All10pcNoise is a short for "Noisy log for the net a12 (a12). The noise type is *mix all* (All) and this log contains at most 10% (10pc) of noise (Noise).". The y-axis contains the values for the analysis metrics.

Additionally, we have plotted the metric values for the mined model and original model with respect to the noisy log and the noise-free one. The reason is that the analysis with the noisy logs allow us to check if the mined models over-fit the data (since these noisy logs were given as input to the genetic algorithms). For instance, if some mined model can proper complete the noisy log (can parse all the traces without missing tokens or tokens left-behind), this mined model has over-fitted the data. On the other hand, when the model does not over-fit the data, the analysis with the noise-free logs can check if the mined model correctly captures the most frequent noise-free behavior (since the noisy logs used for the experiments were created by inserting noisy behavior into these noise-free logs).

In a nutshell, we can conclude that (i) the genetic algorithm is more sensitive to the noise type *swap tasks* (and, consequently, *mix all*) and (ii) the mined models do not tend to over-fit the noisy logs. More specifically, the results point out that:

- The genetic algorithm is more robust to 5% noise than to 10% noise. But the 5% arc post-pruning gave the same results as the 10% one.
- The pruning is more effective for the noise types *missing head*, *missing body*, *missing tail* and *remove task*. This makes sense because these noise types usually can be incorporated to the net by adding causality dependencies to *skip* the "removed" or "missing" tasks. In other words, the main net structure (the one also contained in the original model) does not change, only extra causality dependencies need to be added to it. This explains why the arc post-pruning works quite fine for these noise types.
- Related to the previous item, the noise type *swap tasks* affects the quality of the mined results the most. By looking at figures 25 and 27, one can see that the behavioral/structural precision and recall of the mined models for logs with swapped tasks (`a12Swap5pcNoise` and `a12Swap-10pcNoise`) did not change dramatically after the pruning. This is probably because the



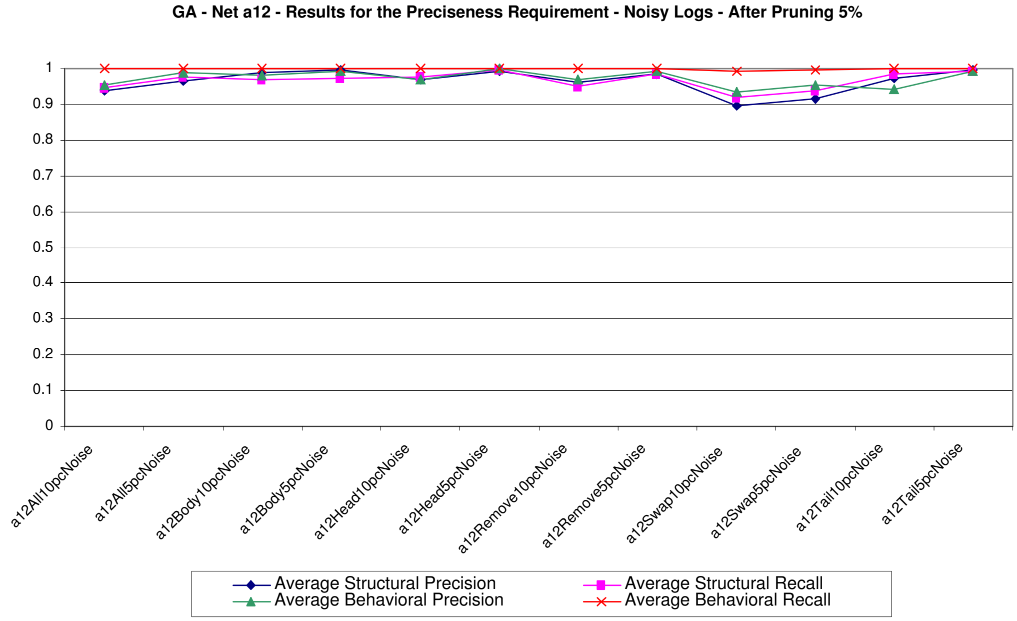


Fig. 25. Average values for the behavioral and structural precision/recall metrics of the models mined by the genetic algorithm (GA) for noisy logs of the net a12. The top (bottom) graph shows the results for the mined models before (after) arc post-pruning. The results show that (i) the mined models have a behavior that is quite similar to the original models (since behavioral precision > 0.8 and behavioral recall > 0.95), and (ii) the arc post-pruning is more effective for the noise types *missing head/body/tail* and *remove task* (since all the values plotted in the bottom graph are better than the ones in the top graph).

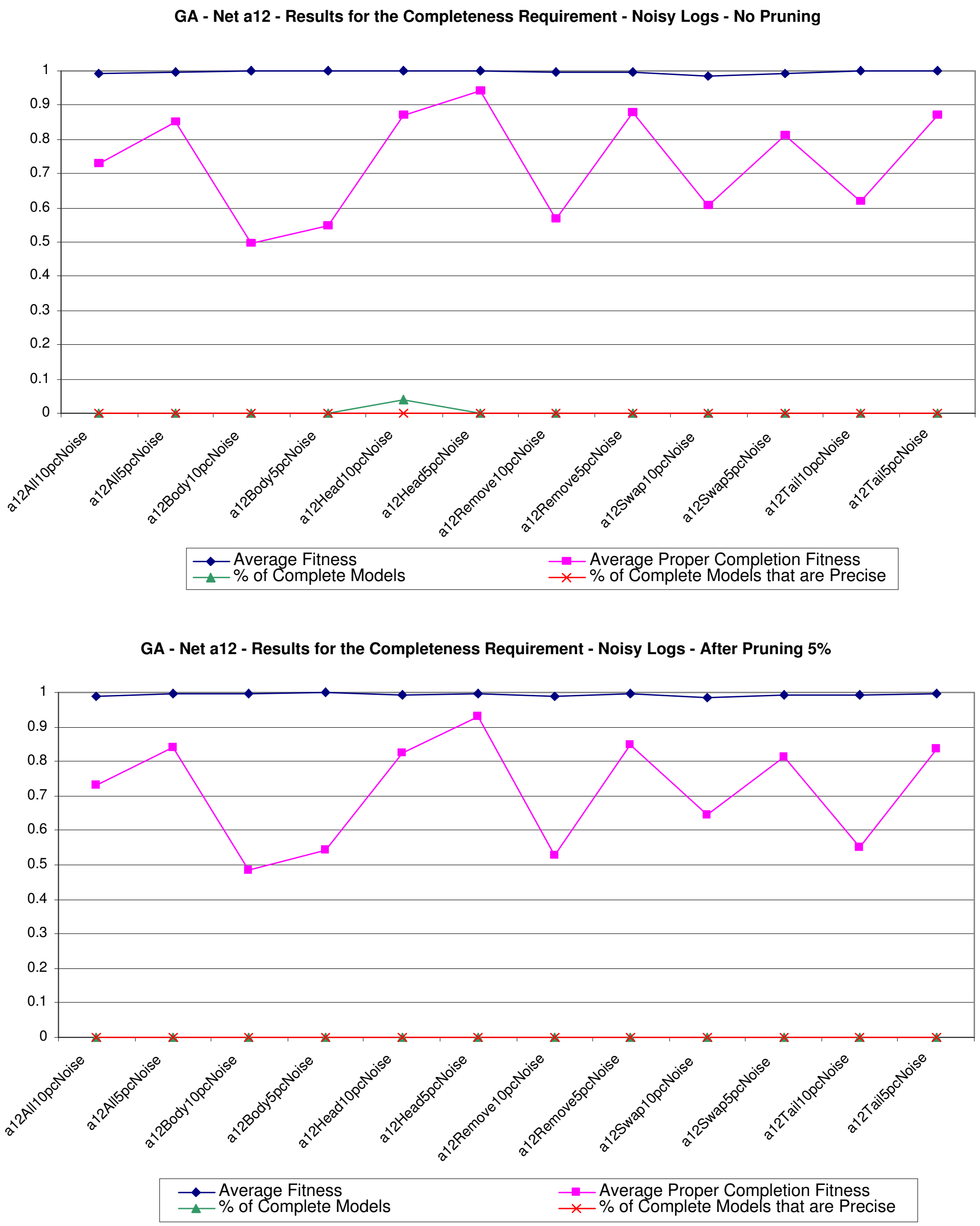
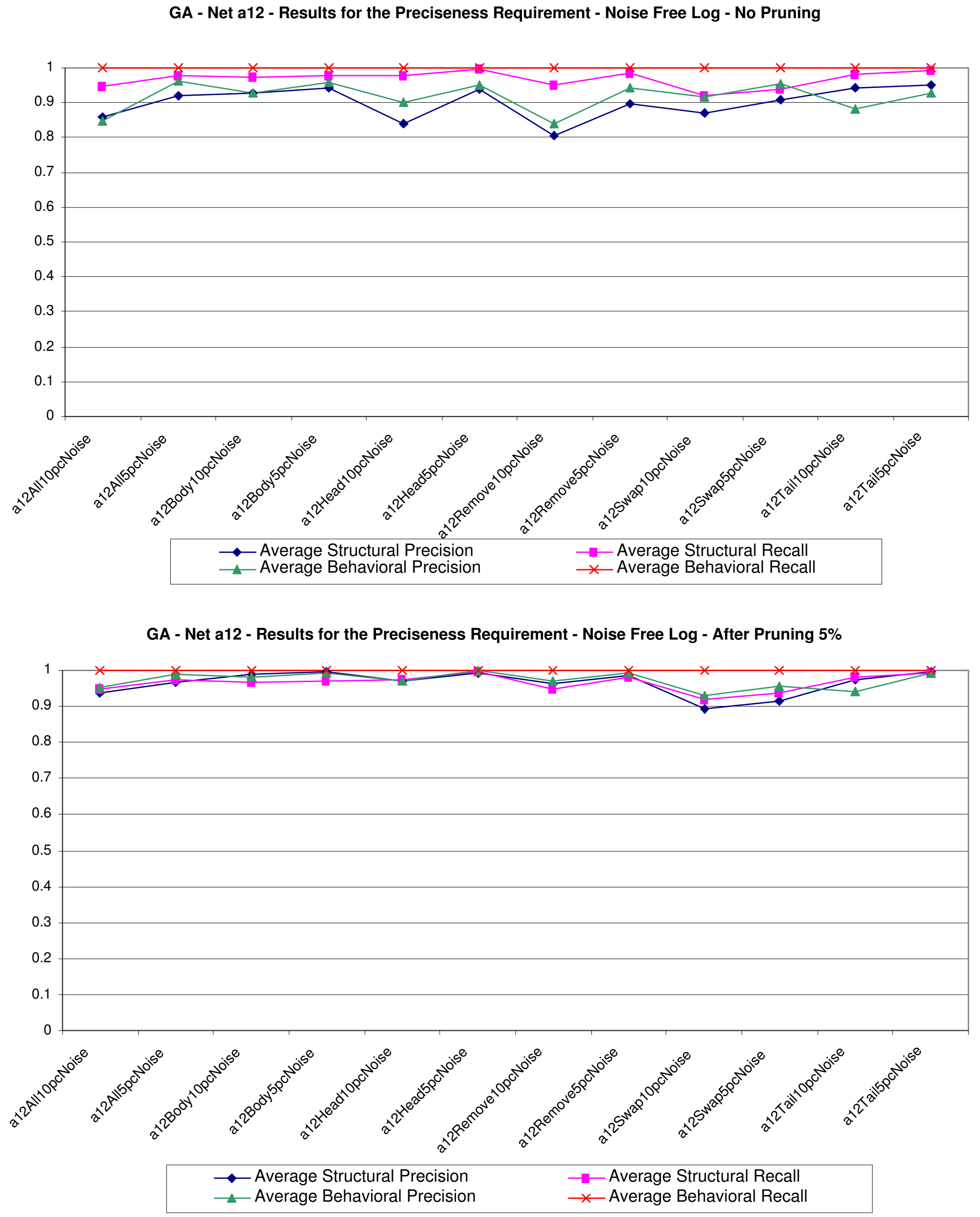


Fig. 26. Results of the GA for the noisy logs of the net a12: completeness metrics. The metrics were calculated based on the noisy logs used during the mining. The top graph shows the results for the mined models. The bottom graph shows the results after the mined models have undergone 5% arc post-pruning. Note that the top graph indicates that only 2 models for the noise type *missing head* (10% noise) over-fit the data. However, overall the mined models did not over-fit the data, since the average proper completion < 0.95 and the % of complete models is 0 for almost all logs.


GA - Net a12 - Results for the Preciseness Requirement - Noise Free Log - No Pruning

GA - Net a12 - Results for the Preciseness Requirement - Noise Free Log - After Pruning 5%

Fig. 27. The results show the same metrics as explained for Figure 25, but these metrics are calculated base on a noise-free log of a12. Note that, contrary to the results in Figure 25, all mined models (before and after pruning) have an average behavioral recall that is equal to 1. This means that, with respect to the noise-free log, all the behavior allowed by the original model is also captured by the mined models.

over-fitting of the mined models to the logs involves more than the simple addition of causality dependencies. I.e., the main structure of mined models is more affected by the *swap tasks* noise type.

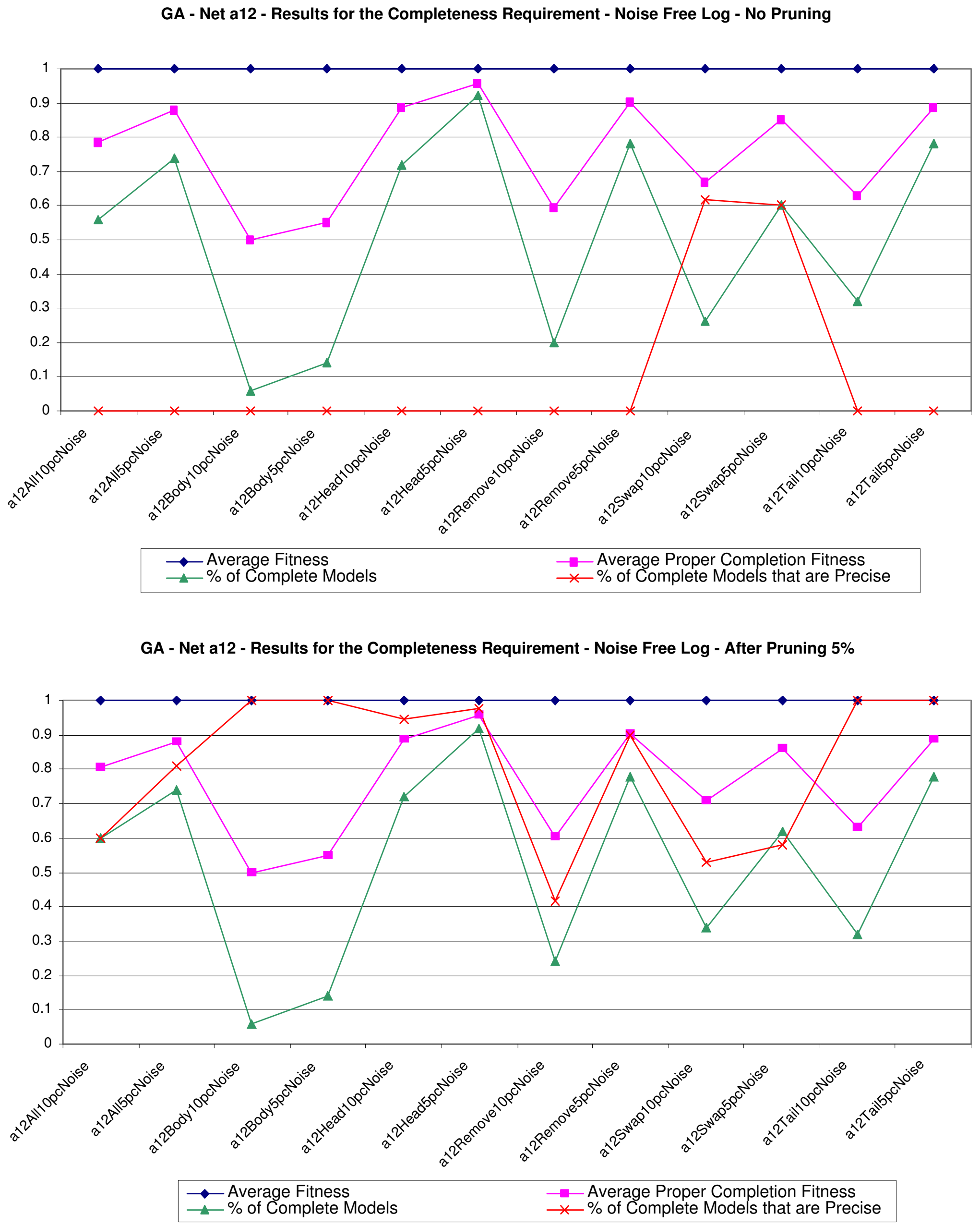


Fig. 28. Same metrics as in Figure 26, but this time the values are calculated based on a noise-free log of a12. The values for the average proper completion fitness point out that the models correctly captured at least 50% (see a12Body10pcNoise) of the behavior in the log. This indicates that the fitness indeed benefits the individuals that correctly model the most frequent behavior in the log. Besides, note that many more mined models are complete and precise after they have undergone arc post-pruning.

The next section shows the results of applying the hybrid version of the genetic algorithm (Scenario IV) to event logs from a municipality in The Netherlands.

7 Mining Real-Life Logs

This section shows the results of applying the genetic algorithm for event logs from a municipality in The Netherlands. The four selected process models deal with the handling of complaints (Bezwaar, BezwaarWOZ, Afschriften) and the process to get a building permit (Bouwvergunning). As shown in Table 2, not all traces in the event logs given to us are compliant with the prescribed (original) process model[20] . So, these logs are a nice setting to test how robust the genetic algorithm is to noise.

Process Model	Number of Traces	% Correctly Parsed Traces
Bezwaar	35	51%
BezwaarWOZ	100	47%
Afschriften	358	100%
Bouwvergunning	407	80%

Table 2
Percentage of traces that could be correctly parsed by the *original* process models. A trace is correctly parsed when no tokens are missing or left behind during the parsing.

The experiments consisted in running the algorithm for every log. The configuration used is just like the one in Section 6.1, but with 10 individuals, at most 5000 generations and for 10 seeds only. For every log, the best mined model over all seeds was selected to undergo the arc post-pruning step (cf. Section 6.2). The arc pruning is a post-processing step that eliminates from a mined model the arcs that are used up to a percentage of the most frequently used arc, during the parsing of the event log by the mined model. The percentage is set by the user. The choice for a post-pruning step is based on the fact that we do not typically know how much noise an event log contains. Thus, we let the genetic algorithm mine the most frequent behavior in the log and use the post-pruning to "clean" this model even further. The results of the experiments are in Table 3. As can be seen, the algorithm indeed goes for the parsing of the most frequent behavior in the log. Note that even after pruning

[20] All modes contain sequences, choices, length-two loops and invisible tasks. Additionally, Bouwvergunning has four tasks in parallel. The amount of tasks per process is: Afschriften (11 tasks), Bezwaar (14), BezwaarWOZ (17) and Bouwvergunning (19).

	% Correctly Parsed Traces Mined Model			
Process Model	no pruning	1% pruning	5% pruning	10% pruning
Bezwaar	100%	100%	62%	51%
BezwaarWOZ	100%	95%	82%	82%
Afschriften	100%	99%	99%	88%
Bouwvergunning	70%	70%	64%	64%

Table 3
Percentage of traces that could be correctly parsed by the *mined* process models before and after the arc pruning. Note that process Bouwvergunning has the only log to which the mined model cannot correctly parse all the traces. This process has a construct with four tasks in parallel. As the results for the experiments in Section 6 indicated, the genetic algorithm goes for the most frequent behavior in these situations.

the arcs that are used 10 or fewer times less than the most frequently used arc, the resulting process model can still correctly parse more than half of the traces in the log. As an example, Figure 29 shows the original and (pruned) mined models for the process BezwaarWOZ (cf. Table 3).

8 Comparison to Some Related Approaches

The genetic algorithm is the only mining algorithm so far (cf. Section 9) that can mine sequence, choice, parallelism, (arbitrary) loops, invisible tasks, and non-free-choice constructs at once, while being robust to noise. Therefore, it is interesting to compare it with other existing approaches. In this section we compare the genetic algorithm with two other mining algorithms: α^{++} and HeuristicsMiner.

The aim of the comparison is to assess if these approaches outperform or not the genetic algorithm when mining logs. The reasons to select these two algorithms are: (i) both algorithms are implemented in the ProM framework (what facilitates the comparison); (ii) the α^{++} [61] is an extension of the α algorithm in [15] to also explicitly capture non-free-choice constructs in the mined model; and (iii) the HeuristicsMiner[21] is robust to noise, can tackle invisible tasks and uses heuristics that are based on the Dependency measure explained in Section 4.1.1. Furthermore, both algorithms can mine the basic structural constructs (sequence, choice, parallelism and loops).

[21] The HeuristicsMiner is the ProM version of the Little Thumb tool [60].

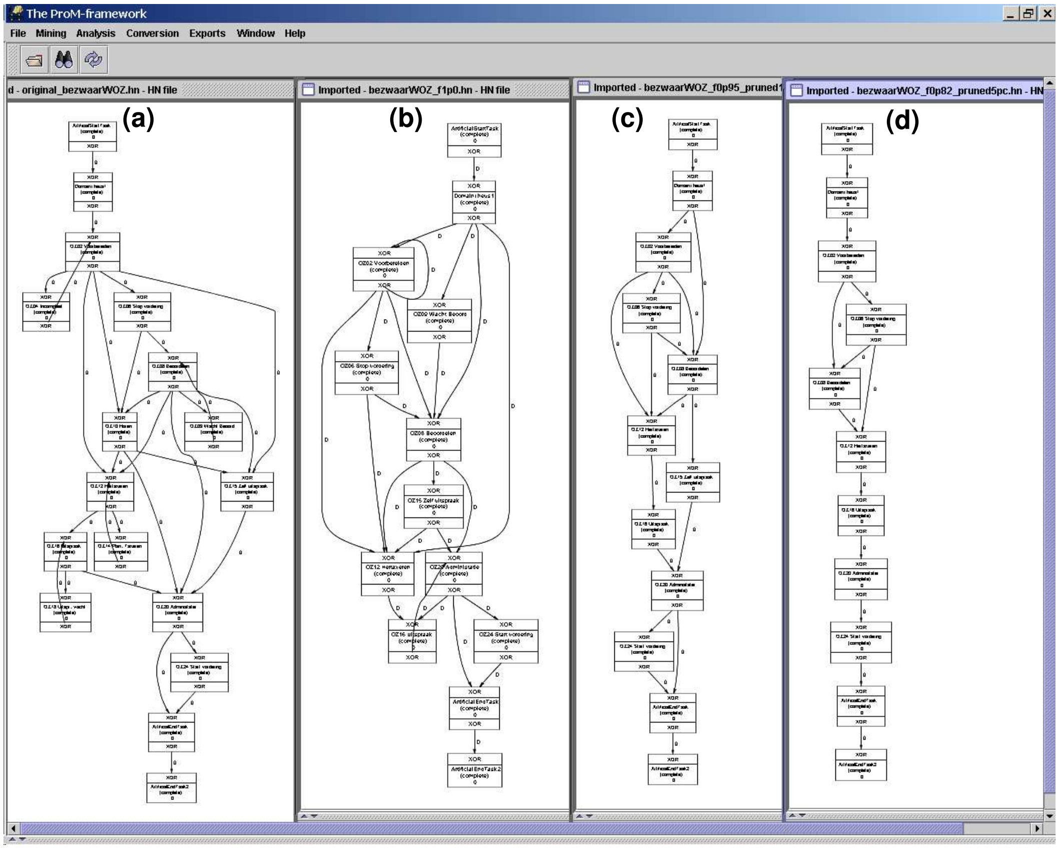

The ProM-framework
File
Mining
Analysis
Conversion
Exports
Window
Help
d - original_bezwaarWOZ.hn - HN file
Imported - bezwaarWOZ_f1p0.hn - HN file
Imported - bezwaarWOZ_f0p95_pruned1
Imported - bezwaarWOZ_f0p82_pruned5pc.hn - HN
(a)
(b)
(c)
(d)
ArtificialStartTask (complete)
Domain heus 1 (complete)
OZ02 Voorbereiden (complete)
OZ09 Wacht Beoord (complete)
OZ06 Stop vordering (complete)
OZ08 Beoordelen (complete)
OZ15 Zelf uitspraak (complete)
OZ12 Hertaxeren (complete)
OZ20 Administratie (complete)
OZ16 uitspraak (complete)
OZ24 Start vordering (complete)
ArtificialEndTask (complete)
ArtificialEndTask2 (complete)
XOR
D

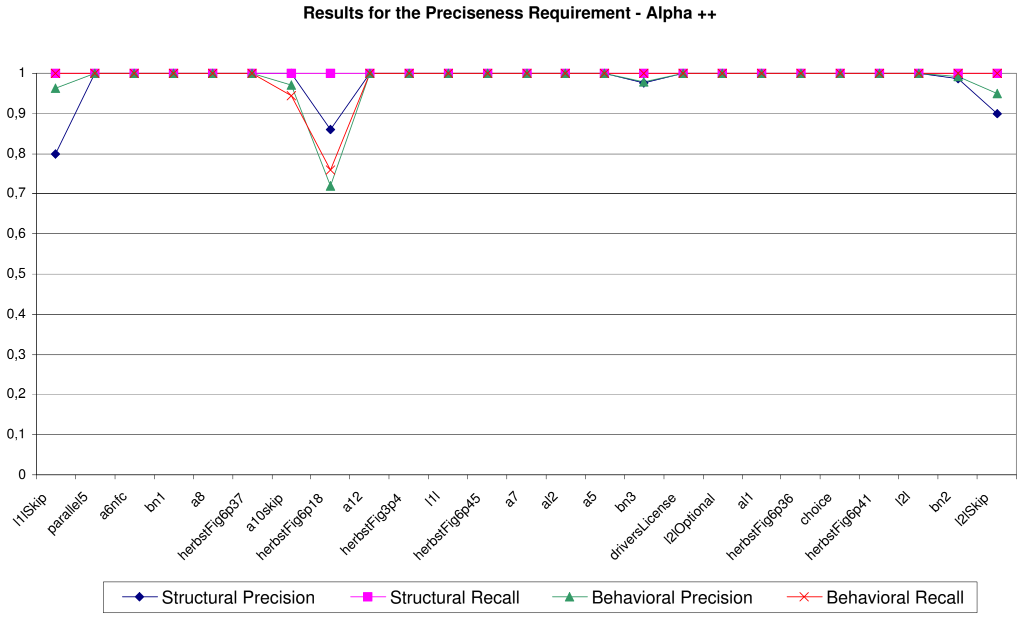
Fig. 29. Original and mined models for the process BezwaarWOZ. The original model is in (a). The unpruned mined model is in (b). (c) and (d) respectively show the models after applying 1% and 5% arc pruning to the mined model in (b).

The α^{++} and HeuristicsMiner algorithms were applied [22] over the three sets of logs (noise-free, noisy and real-life) to which we executed the genetic algorithm. The results show that:

Experiments with noise-free logs: As the values for the analysis metrics (cf. Section 5) in figures 30 and 31 show, both algorithms could find a complete and precise model for many of the logs. However, as expected, the α^{++} could not correctly mine models with invisible tasks (cf. Figure 30) and the HeuristicsMiner was unable to capture non-free-choice constructs (cf. Figure 31).

Experiments with noisy logs: As the α^{++} is not robust to noise, none of its mined models could proper complete any of the traces in the noisy logs in Section 6.2. Actually, the resulting mined models were very spaghetti. However, the HeuristicsMiner outperformed the genetic algorithm for the noisy logs in Section 6.2. In fact, except for the model with non-free-choice (`herbstFig6p36`), the HeuristicMiner mined models with the same struc-

[22] The default configuration provided by ProM was used when running these algorithms.



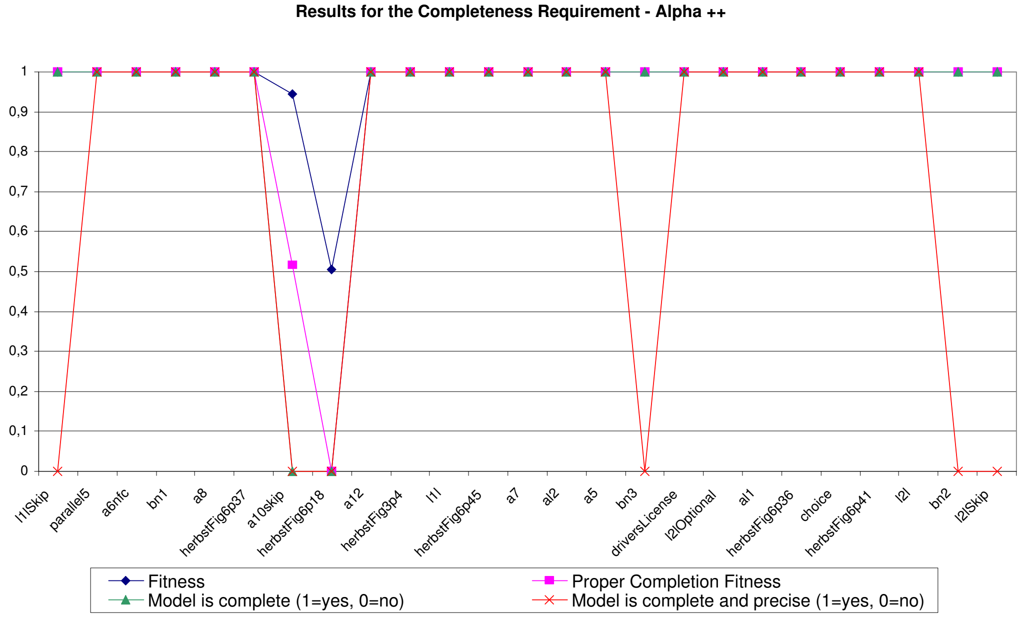


Fig. 30. Results for the models mined by the α^{++} algorithm. Note that the non-free-choice constructs are indeed correctly captured (cf. results for `a6nfc`, `driversLicense` and `herbstFig6p36`), but the algorithm cannot correctly mine the models with invisible tasks (cf. `l1lSkip`, `a10skip`, `herbstFig6p18`, `bn3`, `bn2` and `l2lSkip`).

ture of the original models for all six noise types.

Experiments with real-life logs: Given the results reported in the previous two items, one would expect the HeuristicMiner to outperform the genetic algorithm when running over te real-life logs (cf. Section 7). Especially

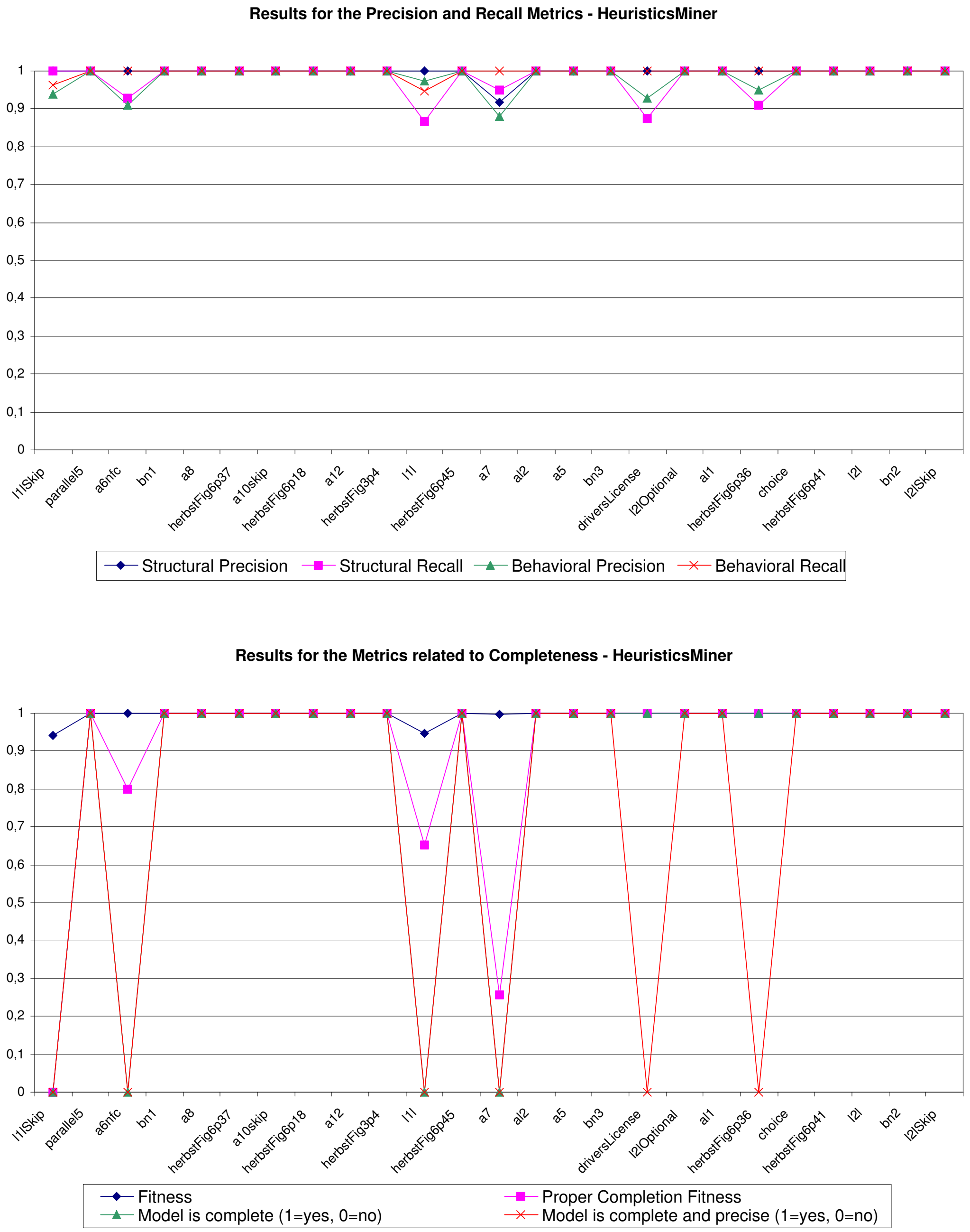


Fig. 31. Results for the models mined by the HeuristicsMiner. Note that the invisible tasks are correctly captured, but the non-free-choice constructs are not. Furthermore, the algorithm seems to have problems with mining length-one loops (cf. `l1lSkip` and `l1l`).

because all the structural constructs in the the prescribed models of the Dutch municipality can be tackled by the HeuristicsMiner. However, as the results in Table 4 indicate, both the α^{++} and the HeuristicsMiner underperformed the genetic algorithm when correctly capturing the most frequent behavior in the log.

	% Correctly Parsed Traces Mined Model	
Process Model	α^{++}	HeuristicsMiner
Bezwaar	0%	48%
BezwaarWOZ	0%	86%
Afschriften	100%	100%
Bouwvergunning	0%	0%

Table 4
Percentage of traces that could be correctly parsed by the models mined by the α^{++} and the HeuristicsMiner. Note that both algorithm underperformed the genetic algorithm (cf. Table 3) when correctly portraying the most frequent behavior in the logs.

As a final remark, we emphasize that both the α^{++} and the HeuristicsMiner are extremely fast (models were mined within 30 seconds on the same computer used for the experiments with the genetic algorithm). However, when looking at the quality of the mined models, the genetic algorithm is better than these other two related algorithms because, for the set of logs discussed above, it mined models that correctly depict the most frequent behavior in more situations.

9 Related Work

The idea of process mining is not new [3,4,6,13,29–32,10–12,37,40,54–56,49,60]. Cook and Wolf have investigated similar issues in the context of software engineering processes. In [11] they describe three methods for process discovery: one using neural networks, one using a purely algorithmic approach, and one Markovian approach. The authors consider the latter two the most promising approaches. The purely algorithmic approach builds a finite state machine where states are fused if their futures (in terms of possible behavior in the next k steps) are identical. The Markovian approach uses a mixture of algorithmic and statistical methods and is able to deal with noise. Note that the results presented in [11] are limited to sequential behavior. Cook and Wolf extend their work to concurrent processes in [10,12]. They propose specific metrics (entropy, event type counts, periodicity, and causality) and use these metrics to discover models out of event streams. However, they do not provide an approach to generate explicit process models. Recall that the final goal of the approach presented in this paper is to find explicit representations for a broad range of process models, i.e., we want to be able to generate a concrete model rather than a set of dependency relations between events. In [13] Cook and Wolf provide a measure to quantify discrepancies between a process model and the actual behavior as registered using event-based data. The idea of applying process mining in the context of workflow management was first introduced in [6]. This work is based on workflow graphs, which are inspired by workflow products such as IBM MQSeries workflow (formerly known as Flowmark) and InConcert. In this paper, two problems are defined. The first problem is to find a workflow graph generating events appearing in a given workflow log. The second problem is to find the definitions of edge conditions. A concrete algorithm is given for tackling the first problem. The approach is quite different from other approaches: Because the nature of workflow graphs there is no need to identify the nature (AND or XOR) of joins and splits. As shown in [35], workflow graphs use true and false tokens which do not allow for cyclic graphs. Nevertheless, [6] partially deals with iteration by enumerating all occurrences of a given activity and then folding the graph. However, the resulting conformal graph is not a complete model. In [40], a tool based on these algorithms is presented. [49] extends the work in [6] to also consider the time information in the logs and, consequently, better detect concurrent behavior. Schimm [54–56] has developed a mining tool suitable for discovering hierarchically structured workflow processes. This requires all splits and joins to be balanced. Herbst and Karagiannis also address the issue of process mining in the context of workflow management [29–32] using an inductive approach. The work presented in [31] is limited to sequential models. The approach described in [30,29,32] also allows for concurrency. It uses stochastic activity graphs as an intermediate representation and it generates a workflow model described in the ADONIS modelling language. In the induction step activity nodes are merged and split in order to discover the underlying process. A notable difference with other approaches is that the same activity can appear multiple times in the workflow model, i.e., *the approach allows for duplicate activities*. The graph generation technique is similar to the approach of [6,40]. The nature of splits and joins (i.e., AND or XOR) is discovered in the transformation step, where the stochastic activity graph is transformed into an ADONIS workflow model with block-structured splits and joins. In contrast to the previous papers, the work [37,60] is characterized by the focus on workflow processes with concurrent behavior (rather than adding ad-hoc mechanisms to capture parallelism). In [60] a heuristic approach using rather simple metrics is used to construct so-called “dependency/frequency tables” and “dependency/frequency graphs”. The preliminary results presented in [60] only provide heuristics and focus on issues such as noise. In [1] the EMiT tool is presented which uses an extended version of the α-algorithm to incorporate timing information. For a detailed description of the α-algorithm and a proof of its correctness we refer to [5]. For a detailed explanation of the constructs the α-algorithm does not correctly mine and an extension to mine short-loops, see [14,15]. The main differences from our work to the above mentioned ones are that (i) our search is not primarily based on local (direct neighborhood) information in the log, (ii) we can capture non-free-choice constructs, and (iii) we try to discover the dependencies (causality relations) and the semantics of the split/join points all together.

With respect to non-free-choice constructs, as already explained in Section 8, Wen et al. [61] have extended the α algorithm in [15] to also discover this kind of construct. The extension is called the α^{++} algorithm. However, unlike the genetic algorithm, the α^{++} is unable to capture invisible tasks and is also not robust to noise. The approach by [26] handles non-free-choice situations by clustering the traces that follow different paths in the mined model. The difference to our approach is that the non-free-choice constructs are not explicitly captured in the control-flow structure of the mined model.

Process mining can be seen as a tool in the context of Business (Process) Intelligence (BPI). In [27] a BPI toolset on top of HP's Process Manager is described. The BPI tools set includes a so-called "BPI Process Mining Engine". However, this engine does not provide any techniques as discussed before. Instead it uses generic mining tools such as SAS Enterprise Miner for the generation of decision trees relating attributes of cases to information about execution paths (e.g., duration). In order to do workflow mining it is convenient to have a so-called "process data warehouse" to store audit trails. Such as data warehouse simplifies and speeds up the queries needed to derive causal relations. In [20,45,46] the design of such warehouse and related issues are discussed in the context of workflow logs. Moreover, [46] describes the PISA tool which can be used to extract performance metrics from workflow logs. Similar diagnostics are provided by the ARIS Process Performance Manager (PPM) [33]. The later tool is commercially available and a customized version of PPM is the Staffware Process Monitor (SPM) [57] which is tailored towards mining Staffware logs. Note that none of the latter tools is extracting the process model. The main focus is on clustering and performance analysis rather than causal relations as in [6,11–13,29–31,37,40,54,55,60].

More from a theoretical point of view, the rediscovery problem discussed in this paper is related to the work discussed in [7,23,50]. In these papers the limits of inductive inference are explored. For example, in [23] it is shown that the computational problem of finding a minimum finite-state acceptor compatible with given data is NP-hard. Several of the more generic concepts discussed in these papers could be translated to the domain of process mining. It is possible to interpret the problem described in this paper as an inductive inference problem specified in terms of rules, a hypothesis space, examples, and criteria for successful inference. The comparison with literature in this domain raises interesting questions for process mining, e.g., how to deal with negative examples (i.e., suppose that besides log W there is a log V of traces that are not possible, e.g., added by a domain expert). However, despite the many relations with the work described in [7,23,50] there are also many differences, e.g., we are mining at the net level rather than sequential or lower level representations (e.g., Markov chains, finite state machines, or regular expressions). For a survey of existing research, we also refer to [3].

There have been some papers combining Petri nets and genetic algorithms, cf. [8,34,38,39,42–44,48,51,58]. However, these papers do not try to discover a process model based on some event log. The approach in this paper is the first approach using genetic algorithms for process discovery. The goal of using genetic algorithms is to tackle problems such as duplicate activities, hidden activities, non-free-choice constructs, noise, and incompleteness, i.e., overcome the problems of some of the traditional approaches. Actually, we have also previous papers on genetic process mining [17,59]. However, the work presented in this paper differs from our previous papers for the following reasons. In [59], the causal matrix representation (cf. Definition 5) was more restrictive because the subsets in the input and output condition functions (i.e., I and O) were *partition sets* of A. Thus, nets like the one in Figure 5 could not be supported. The work in [17] removed these restrictions and improved the fitness measure. However, neither the fitness measure in [59] nor the one in [17] have the preciseness requirement (cf. Section 4.2.2) to punish over-general models. Additionally, the works in [17,59] (i) do not include the analysis metrics (cf. Section 5) to assess the quality of the mined models, (ii) have experiments with a smaller set of synthetic logs, and (iii) do not include experiments with real-life logs.

With respect to the analysis metrics, some notions related to the behavioral precision metric (cf. Definition 13) have been used in [26,53]. In [26], the notion of *soundness* is used to check for how much extra behavior a mined model allows for. However, the metric assumes that the target model does not contain loops. In [53], the notion of *behavioral appropriateness* is defined to also check if a model can generate more behavior than the one expressed in the log. The problem here is that the metric cannot state when a model is precise enough. Because the analysis metrics defined in Section 5 make use of the original model as well, they can precisely quantify how complete and precise a mined model is.

10 Conclusions and Future Work

In this paper we have presented a hybrid genetic algorithm to mine process models from event logs. The internal representation (the causal matrix) supports more complex routing constructs like non-free choice and invisible task. The fitness measure benefits the individuals that are complete (can parse most of the behavior in the log) and precise (cannot parse more behavior than the one that can be derived from the log). The genetic operators (crossover and mutation) manipulate the basic genetic material in the algorithm: the causality relations. The experiments with synthetic logs show that (i) the use of heuristics is indeed beneficial to speed the search performed by the genetic algorithm and (ii) the genetic operators are playing their role in finding the *non-local* causality relations that can never be introduced by the heuristics. Furthermore, the experiments with synthetic noisy logs and real-life logs show that the genetic algorithm is capturing the most frequent behavior in the log even in the presence of noise. However, although the GA is able to mine models with all structural constructs but duplicates tasks and is robust to noise, it has a drawback that cannot be neglected: the computational time. For this reason, future work will focus on developing better strategies to perform this search. Additionally, we want to develop a genetic algorithm that can also mine process models with duplicate tasks. As a final remark, the genetic algorithm, analysis metrics and arc post-pruning step are all implemented as plug-ins in the ProM framework (www.processmining.org).

References

[1] W.M.P. van der Aalst and B.F. van Dongen. Discovering Workflow Performance Models from Timed Logs. In Y. Han, S. Tai, and D. Wikarski, editors, *International Conference on Engineering and Deployment of Cooperative Information Systems (EDCIS 2002)*, volume 2480 of *Lecture Notes in Computer Science*, pages 45–63. Springer-Verlag, Berlin, 2002.

[2] W.M.P. van der Aalst and M. Song. Mining Social Networks: Uncovering interaction patterns in business processes. In J. Desel, B. Pernici, and M. Weske, editors, *International Conference on Business Process Management (BPM 2004)*, volume 3080 of *Lecture Notes in Computer Science*, pages 244–260. Springer-Verlag, Berlin, 2004.

[3] W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.

[4] W.M.P. van der Aalst and A.J.M.M. Weijters, editors. *Process Mining*, volume 53 of *Special Issue of Computers in Industry*. Elsevier Science Publishers, Amsterdam, 2004.

[5] W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[6] R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In I. Ramos G. Alonso H.-J. Schek, F. Saltor, editor, *Advances in Database Technology - EDBT'98: Sixth International Conference on Extending Database Technology*, volume 1377 of *Lecture Notes in Computer Science*, pages 469–483, 1998.

[7] D. Angluin and C.H. Smith. Inductive Inference: Theory and Methods. *Computing Surveys*, 15(3):237–269, 1983.

[8] T. Bourdeaud'huy and P. Yim. Petri net controller synthesis using genetic search. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, volume 1, pages 528–533, 2002.

[9] E. Cantú-Paz, J. A. Foster, K. Deb, L. Davis, R. Roy, U. O'Reilly, H. Beyer, R. K. Standish, G. Kendall, S. W. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. A. Dowsland, N. Jonoska, and J. F. Miller, editors. *Genetic and Evolutionary Computation - GECCO 2003, Genetic and Evolutionary Computation Conference, Chicago, IL, USA, July 12-16, 2003. Proceedings, Part II*, volume 2724 of *Lecture Notes in Computer Science*. Springer, 2003.

[10] J.E. Cook, Z. Du, C. Liu, and A.L. Wolf. Discovering Models of Behavior for Concurrent Workflows. *Computers in Industry*, 53(3):297–319, 2004.

[11] J.E. Cook and A.L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[12] J.E. Cook and A.L. Wolf. Event-Based Detection of Concurrency. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, pages 35–45, New York, NY, USA, 1998. ACM Press.

[13] J.E. Cook and A.L. Wolf. Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model. *ACM Transactions on Software Engineering and Methodology*, 8(2):147–176, 1999.

[14] A.K. Alves de Medeiros, W.M.P. van der Aalst, and A.J.M.M. Weijters. Workflow Mining: Current Status and Future Directions. In R. Meersman, Z. Tari, and D.C. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 389–406. Springer-Verlag, Berlin, 2003.

[15] A.K. Alves de Medeiros, B.F. van Dongen, W.M.P. van der Aalst, and A.J.M.M. Weijters. Process Mining: Extending the α-algorithm to Mine Short Loops. BETA Working Paper Series, WP 113, Eindhoven University of Technology, Eindhoven, 2004.

[16] A.K. Alves de Medeiros, A.J.M.M. Weijters, and W.M.P. van der Aalst. Using Genetic Algorithms to Mine Process Models: Representation, Operators and Results. BETA Working Paper Series, WP 124, Eindhoven University of Technology, Eindhoven, 2004.

[17] A.K. Alves de Medeiros, A.J.M.M. Weijters, and W.M.P. van der Aalst. Genetic Process Mining: A Basic Approach and its Challenges. In *Busines Process Management 2005 Workshops*, volume 3812 of *Lecture Notes in Computer Science*, pages 203–215. Springer Verlag, 2006.

[18] J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.

[19] M. Dumas, W.M.P. van der Aalst, and A.H. ter Hofstede, editors. *Process-Aware Information Systems: Bridging People and Software Through Process Technology*. John Wiley & Sons Inc, 2005.

[20] J. Eder, G.E. Olivotto, and Wolfgang Gruber. A Data Warehouse for Workflow Logs. In Y. Han, S. Tai, and D. Wikarski, editors, *International Conference on Engineering and Deployment of Cooperative Information Systems (EDCIS 2002)*, volume 2480 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, Berlin, 2002.

[21] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Natural Computing. Springer-Verlag, Berlin, 2003.

[22] R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[23] E.M. Gold. Complexity of Automaton Identification from Given Data. *Information and Control*, 37(3):302–320, 1978.

[24] G. Greco, A. Guzzo, and L. Pontieri. Mining Hierarchies of Models: From Abstract Views to Concrete Specifications. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Business Process Management*, volume 3649, pages 32–47, 2005.

[25] G. Greco, A. Guzzo, L. Pontieri, and D. Saccà. Mining Expressive Process Models by Clustering Workflow Traces. In H. Dai, R. Srikant, and C. Zhang, editors, *PAKDD*, volume 3056 of *Lecture Notes in Computer Science*, pages 52–62. Springer, 2004.

[26] G. Greco, A. Guzzo, L. Pontieri, and D. Sacca. Discovering expressive process models by clustering log traces. *IEEE Transactions on Knowledge and Data Engineering*, 18(8):1010–1027, 2006.

[27] D. Grigori, F. Casati, U. Dayal, and M.C. Shan. Improving Business Process Quality through Exception Understanding, Prediction, and Prevention. In P. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. Snodgrass, editors, *Proceedings of 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 159–168. Morgan Kaufmann, 2001.

[28] P. D. Grunwald, I. J. Myung, and M. Pitt, editors. *Advances in Minimum Description Length Theory and Applications*. The MIT Press, 2005.

[29] J. Herbst. Dealing with Concurrency in Workflow Induction. In U. Baake, R. Zobel, and M. Al-Akaidi, editors, *European Concurrent Engineering Conference*. SCS Europe, 2000.

[30] J. Herbst. *Ein induktiver Ansatz zur Akquisition und Adaption von Workflow-Modellen*. PhD thesis, Universität Ulm, November 2001.

[31] J. Herbst and D. Karagiannis. Integrating Machine Learning and Workflow Management to Support Acquisition and Adaptation of Workflow Models. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 9:67–92, 2000.

[32] J. Herbst and D. Karagiannis. Workflow Mining with InWoLvE. *Computers in Industry*, 53(3):245–264, 2004.

[33] IDS Scheer. ARIS Process Performance Manager (ARIS PPM). http://www.ids-scheer.com, 2002.

[34] S. Malpathak K. Saitou and H. Qvam. Robust design of flexible manufacturing systems using, colored petri net and genetic algorithm. *Journal of Intelligent Manufacturing*, 13(5):339–351, 2002.

[35] B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows (submitted)*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2002. Available via http://www.tm.tue.nl/it/research/patterns.

[36] L. Maruster. *A Machine Learning Approach to Understand Business Processes*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2003.

[37] L. Maruster, A.J.M.M. Weijters, W.M.P. van der Aalst, and A. van den Bosch. Process Mining: Discovering Direct Successors in Process Logs. In *Proceedings of the 5th International Conference on Discovery Science (Discovery Science 2002)*, volume 2534 of *Lecture Notes in Artificial Intelligence*, pages 364–373. Springer-Verlag, Berlin, 2002.

[38] H. Mauch. Evolving Petri Nets with a Genetic Algorithm. In E. Cantu-Paz and J.A. Foster et al., editors, *Genetic and Evolutionary Computation (GECCO 2003)*, volume 2724 of *Lecture Notes in Computer Science*, pages 1810–1811. Springer-Verlag, Berlin, 2003.

[39] Holger Mauch. Evolving petri nets with a genetic algorithm. In Cantú-Paz et al. [9], pages 1810–1811.

[40] M.K. Maxeiner, K. Küspert, and F. Leymann. Data Mining von Workflow-Protokollen zur teilautomatisierten Konstruktion von Prozemodellen. In *Proceedings of Datenbanksysteme in Büro, Technik und Wissenschaft*, pages 75–84. Informatik Aktuell Springer, Berlin, Germany, 2001.

[41] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–77, 1992.

[42] J. H. Moore and L. W. Hahn. An Improved Grammatical Evolution Strategy for Hierarchical Petri Net Modeling of Complex Genetic Systems. In G. R. Raidl et al., editor, *Applications of Evolutionary Computing, EvoWorkshops2004*, volume 3005 of *Lecture Notes in Computer Science*, pages 63–72. Springer-Verlag, Berlin, 2004.

[43] J. H. Moore and L. W. Hahn. Grammatical evolution for the discovery of petri net models of complex genetic systems. In Cantú-Paz et al. [9], pages 2412–2413.

[44] J. H. Moore and L. W. Hahn. Petri net modeling of high-order genetic systems using grammatical evolution. *BioSystems*, 2003.

[45] M. zur Mühlen. Process-driven Management Information Systems Combining Data Warehouses and Workflow Technology. In B. Gavish, editor, *Proceedings of the International Conference on Electronic Commerce Research (ICECR-4)*, pages 550–566. IEEE Computer Society Press, Los Alamitos, California, 2001.

[46] M. zur Mühlen and M. Rosemann. Workflow-based Process Monitoring and Controlling - Technical and Organizational Issues. In R. Sprague, editor, *Proceedings of the 33rd Hawaii International Conference on System Science (HICSS-33)*, pages 1–10. IEEE Computer Society Press, Los Alamitos, California, 2000.

[47] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[48] J. Nummela and B. A. Julstrom. Evolving petri nets to represent metabolic pathways. In H. Beyer and U. O'Reilly, editors, *GECCO*, pages 2133–2139. ACM, 2005.

[49] S.S. Pinter and M. Golani. Discovering Workflow Models from Activities Lifespans. *Computers in Industry*, 53(3):283–296, 2004.

[50] L. Pitt. Inductive Inference, DFAs, and Computational Complexity. In K.P. Jantke, editor, *Proceedings of International Workshop on Analogical and Inductive Inference (AII)*, volume 397 of *Lecture Notes in Computer Science*, pages 18–44. Springer-Verlag, Berlin, 1889.

[51] J.P. Reddy, S. Kumanan, and O.V.K. Chetty. Application of Petri Nets and a Genetic Algorithm to Multi-Mode Multi-Resource Constrained Project Scheduling. *International Journal of Advanced Manufacturing Technology*, 17(4):305–314, 2001.

[52] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.

[53] A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In Christoph Bussler and Armin Haller, editors, *Business Process Management Workshops*, volume 3812, pages 163–176, 2005.

[54] G. Schimm. Process Mining. http://www.processmining.de/.

[55] G. Schimm. Process Miner - A Tool for Mining Process Schemes from Event-based Data. In S. Flesca and G. Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA)*, volume 2424 of *Lecture Notes in Computer Science*, pages 525–528. Springer-Verlag, Berlin, 2002.

[56] G. Schimm. Mining Exact Models of Concurrent Workflows. *Computers in Industry*, 53(3):265–281, 2004.

[57] Staffware. Staffware Process Monitor (SPM). http://www.staffware.com, 2002.

[58] H. Tohme, M. Nakamura, E. Hachiman, and K. Onaga. Evolutionary petri net approach to periodic job-shop-scheduling. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, volume 4, pages 441–446, 1999.

[59] W.M.P. van der Aalst, A.K. Alves de Medeiros, and A.J.M.M. Weijters. Genetic Process Mining. In *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets*, volume 3536 of *Lecture Notes in Computer Science*, 2005.

[60] A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.

[61] L. Wen, J. Wang, and J. Sun. Detecting Implicit Dependencies Between Tasks from Event Logs. In Xiaofang Zhou, Jianzhong Li, Heng Tao Shen, Masaru Kitsuregawa, and Yanchun Zhang, editors, *APWeb*, volume 3841 of *Lecture Notes in Computer Science*, pages 591–603. Springer, 2006.

A Details About the Models for Experiments with Synthetic Logs

Net	Sequence	Choice	Parallelism	Length-One Loop	Length-Two Loop	Structured Loop	Arbitrary Loop	Non-Local NFC	Invisible Tasks	Duplicates in Sequence	Duplicates in Parallel	Number of Visible Tasks
a10skip	✓	✓	✓						✓			12
a12	✓	✓	✓									14
a5	✓	✓	✓	✓								7
a6nfc	✓	✓	✓					✓				8
a7	✓	✓	✓									9
a8	✓	✓	✓									10
al1	✓	✓	✓				✓					9
al2	✓	✓	✓				✓					13
bn1	✓	✓										42
bn2	✓	✓				✓			✓			42
bn3	✓	✓				✓			✓			42
choice	✓	✓										12
driversLicense	✓	✓						✓				9
herbstFig3p4	✓	✓	✓			✓						12
herbstFig6p18	✓	✓		✓	✓	✓			✓			7
herbstFig6p36	✓	✓						✓				12
herbstFig6p37	✓		✓									16
herbstFig6p41	✓	✓	✓									16
herbstFig6p45	✓		✓									8
l1l	✓	✓		✓								6
l1lSkip	✓	✓	✓	✓					✓			6
l2l	✓	✓			✓							6

Continued on next page

Net	Sequence	Choice	Parallelism	Length-One Loop	Length-Two Loop	Structured Loop	Arbitrary Loop	Non-Local NFC	Invisible Tasks	Duplicates in Sequence	Duplicates in Parallel	Number of Visible Tasks
l2lOptional	✓	✓			✓							6
l2lSkip	✓	✓			✓				✓			6
parallel5	✓		✓									10

Table A.1: Overview of the structural constructs contained in the models for the experiments with synthetic logs.