Using Genetic Algorithms to Mine Process Models: Representation, Operators and Results

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.
{a.k.medeiros, a.j.m.m.weijters, w.m.p.v.d.aalst}@tm.tue.nl

Abstract. The topic of process mining has attracted the attention of both researchers and tool vendors in the Business Process Management (BPM) space. The goal of process mining is to *discover* process models from event logs, i.e., events logged by some information system are used to extract information about activities and their causal relations. Several algorithms have been proposed for process mining. Many of these algorithms cannot deal with concurrency. Other typical problems are the presence of duplicate activities, hidden activities, non-free-choice constructs, etc. In addition, real-life logs contain noise (e.g., exceptions or incorrectly logged events) and are typically incomplete (i.e., the event logs contain only a fragment of all possible behaviors). To tackle these problems we propose a completely new approach based on genetic algorithms. In this paper, we present a new process representation, a fitness measure and the genetic operators used in a genetic algorithm to mine process models. Our focus is on the use of the genetic algorithm for mining noisy event logs. Additionally, in the appendix we elaborate on the relation between Petri nets and this representation and show that genetic algorithms can be used to discover Petri net models from event logs.

Keywords: process mining, genetic mining, genetic algorithms.

1 Introduction

Buzzwords such as Business Process Intelligence (BPI) and Business Activity Monitoring (BAM) illustrate the practical interest in techniques to extract knowledge from the information recorded by today's information systems. Most information systems support some form of logging. For example, Enterprise Resource Planning (ERP) systems such as SAP R/3, PeopleSoft, Oracle, JD Edwards, etc. log transactions at various levels. Any Workflow Management (WfM) system records audit trails for individual cases. The Sarbanes-Oxley act is forcing organizations to log even more information. The availability of this information triggered the need for process mining techniques that analyze event logs.

The goal of process mining is to extract information about processes from transaction logs [5]. We assume that it is possible to record events such that (i) each event refers to an *activity* (i.e., a well-defined step in the process), (ii) each event refers to a *case* (i.e., a process instance), (iii) each event *can* have a *performer* also referred to as *originator* (the actor executing or initiating the activity), and (iv) events *can* have a *timestamp* and are totally ordered. Table 1 shows an example of a log involving 18 events and 8 activities. In addition to

the information shown in this table, some event logs contain more information on the case itself, i.e., data elements referring to properties of the case.

case id	activity id	originator	timestamp
case 1	activity A	John	9-3-2004:15.01
case 2	activity A	John	9-3-2004:15.12
case 3	activity A	Sue	9-3-2004:16.03
case 3	activity D	Carol	9-3-2004:16.07
case 1	activity B	Mike	9-3-2004:18.25
case 1	activity H	John	10-3-2004:9.23
case 2	activity C	Mike	10-3-2004:10.34
case 4	activity A	Sue	10-3-2004:10.35
case 2	activity H	John	10-3-2004:12.34
case 3	activity E	Pete	10-3-2004:12.50
case 3	activity F	Carol	11-3-2004:10.12
case 4	activity D	Pete	11-3-2004:10.14
case 3	activity G	Sue	11-3-2004:10.44
case 3	activity H	Pete	11-3-2004:11.03
case 4	activity F	Sue	11-3-2004:11.18
case 4	activity E	Clare	11-3-2004:12.22
case 4	activity G	Mike	11-3-2004:14.34
case 4	activity H	Clare	11-3-2004:14.38

Table 1. An event log (audit trail).

Event logs such as the one shown in Table 1 are used as the starting point for mining. We distinguish three different mining perspectives: (1) the process perspective, (2) the organizational perspective and (3) the case perspective. The *process perspective* focuses on the control-flow, i.e., the ordering of activities. The goal of mining this perspective is to find a good characterization of all possible paths, expressed in terms of a process model (e.g., expressed in terms of a Petri net [38] or Event-driven Process Chain (EPC) [23, 24]). The *organizational perspective* focuses on the originator field, i.e., which performers are involved and how they are related. The goal is to either structure the organization by classifying people in terms of roles and organizational units or to show relation between individual performers (i.e., build a social network [4]). The *case perspective* focuses on properties of cases. Cases can be characterized by their path in the process or by the originators working on a case. However, cases can also be characterized by the values of the corresponding data elements. For example, if a case represents a replenishment order it is interesting to know if delayed orders have common properties. The process perspective is concerned with the "How?" question, the organizational perspective is concerned with the "Who?" question, and the case perspective is concerned with the "What?" question. In this paper we will focus completely on the process perspective, i.e., the ordering of the activities. This means that here we ignore the last two columns in Table 1. For the mining of the other perspectives we refer to [5] and http://www.processmining.org.

Note that the ProM tool described in this paper is able to mine the other perspectives and can also deal with other issues such as transactions, e.g., in

the ProM tool we consider different event types such as "schedule", "start", "complete", "abort", etc. However, for reasons of simplicity we abstract from this in this paper and consider activities to be atomic as shown in Table 1.

If we abstract from the other perspectives, Table 1 contains the following information: case 1 has event trace A, B, H, case 2 has event trace A, C, H, case 3 has event trace A, D, E, F, G, H, and case 4 has event trace A, D, F, E, G, H. If we analyze these four sequences we can extract the following information about the process (assuming some notion of completeness and no noise). The underlying process has 8 activities (A, B, C, D, E, F, G and H). A is always the first activity to be executed and H is always the last one. After A is executed, activities B, C *or* D can be executed. In other words, after A, there is a *choice* in the process and only one of these activities can be executed next. When B or C are executed, they are followed by the execution of H (see cases 1 and 2). When D is executed, both E *and* F can be executed in any order. Since we do not consider explicit parallelism, we assume E and F to be concurrent (see cases 3 and 4). Activity G synchronizes the parallel branches that contain E and F. Activity H is executed whenever B, C *or* G has been executed. We can use a Petri net [38] as shown in Figure 1 to model the four cases of the event log in Table 1.

Petri nets are a formalism to model concurrent processes. Graphically, Petri nets are bipartite directed graphs with two node types: *places* and *transitions*. The places represent conditions in the process. The transitions represent actions. The activities 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 1, the process' initial state has only one token in place *Start*. This means that A is the only transition that can be executed in the initial state. When A executes (or fires), one token is removed from the place *Start* and one token is added to the place $p1$.

The Petri net shown in Figure 1 is a good model for the event log containing the four cases. Note that each of the four cases can be "reproduced" by the Petri net shown in Figure 1, i.e. the Petri net contains all observed behavior. In this case, all possible firing sequences of the Petri net shown in Figure 1 are contained in the log. Generally, this is not the case since in practice it is unrealistic to assume that all possible behavior is always contained in the log, cf. the discussion on completeness in [7].

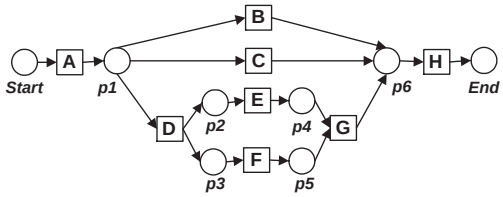


Fig. 1. Petri net discovered based on the event log in Table 1.

Existing approaches for mining the process perspective [5, 7, 8, 10, 19, 29, 42] have problems dealing with issues such as duplicate activities, hidden activities, non-free-choice constructs, noise, and incompleteness. The problem with *duplicate activities* occurs when the same activity can occur at multiple places in the process. This is a problem because it is no longer clear to which activity some event refers. The problem with *hidden activities* is that essential routing decisions are not logged but impact the routing of cases. *Non-free-choice* constructs are problematic because it is not possible to separate choice from synchronization. We consider two sources of *noise*: (1) incorrectly logged events (i.e., the log does not reflect reality) or (2) exceptions (i.e., sequences of events corresponding to "abnormal behavior"). Clearly noise is difficult to handle. The problem of *incompleteness* is that for many processes it is not realistic to assume that all possible behavior is contained in the log. For processes with many alternative routes and parallelism, the number of possible event traces is typically exponential in the number of activities, e.g., a process with 10 binary choices in a sequence will have 1024 possible event sequences and a process with 10 activities in parallel will have even 3628800 possible event sequences. In this paper we focus on noise and incompleteness.

We can consider process mining as a search for the most appropriate process out of the search space of candidate process models. Mining algorithms can use different strategies to find the most appropriate model. Two extreme strategies can be distinguished (i) *local strategies* primarily based on a step by step building of the optimal process model based on very local information, and (ii) *global strategies* primarily based on an one strike search for the optimal model. Most process mining approaches use a local strategy. An example of an algorithm using a local strategy is the α-algorithm [7] where only very local information about binary relations between events is used. A genetic search is an example of a very global search strategy; because the quality or fitness of a candidate model is calculated by comparing the process model with all traces in the event log the search process becomes very global. For local strategies there is no guarantee that the outcome of the locally optimal steps (at the level of binary event relations) will result in a globally optimal process model. Hence, the performance of local mining techniques can be seriously hampered when the necessary information is not locally available (e.g. one erroneous example can completely mess up the derivation of a right model). Therefore, we started to use *Genetic Algorithms* (GA).

In this paper, we present a genetic algorithm to discover a Petri net given a set of event traces. Genetic algorithms are adaptive search methods that try

to mimic the process of evolution [15, 31]. These algorithms start with an initial population of individuals (in this case process models). Populations evolve by selecting the fittest individuals and generating new individuals using genetic operations such as crossover (combining parts of two of more individuals) and mutation (random modification of an individual). Our initial experiences showed that a representation of individuals in terms of a Petri net is not a very convenient. First of all, the Petri net contains places that are not visible in the log. Note that in Figure 1 we cannot assign meaningful names to places. Second, the classical Petri net is not very convenient notation for generating an initial population because it is difficult to apply simple heuristics. Third, the definition of the genetic operators (crossover and mutation) is cumbersome. Finally, the expressive power of Petri nets is in some cases too limited (combinations of AND/OR-splits/joins). Therefore, we use an new representation named *casual matrix*.

The remainder of this paper is organized as follows. Section 2 describes the process representation used in our GA approach. Section 3 explains the details of the GA (i.e. the initialization process, the fitness measure, and the crossover and mutation operations). Section 4 discusses the experimental results. Section 5 discusses some related work. Section 6 has the conclusions and future work. For the readers familiar with Petri nets, Appendix A explains and formalizes the relation between the causal matrix and Petri nets.

2 Internal Representation

In this section we first explain the *causal matrix* that we use to encode individuals (i.e. processes) in our genetic population. After that we discuss the semantics of causal matrices.

A process model describes the *routing of activities* for a given business process. The routing shows which activities are a direct cause for other activities. When an activity is the single cause of another activity, there is a *sequential* routing (see Figure 2 - sequence). When an activity enables the execution of multiple concurrent activities, there is a *parallel* routing (see Figure 2 - parallelism). When an activity enables the execution of multiple activities but only one of these activities can actually be executed, there is a *choice* routing (see Figure 2 - choice). Note that the basic routing constructs sequence, parallelism and choice can be combined to model more complex ones (for instance, a loop can be seen as the combination of a sequence and a choice where the OR-join precedes the OR-split.). Given these observations about routing constructs, a process model must express (i) the process' activities, (ii) which activities cause/enable others, and (iii) if the causal relation between activities are combined in a sequential, parallel or choice routing.

2.1 Causal Matrices

A process model is *conceptually* a matrix with boolean expressions associated to its rows and columns. The matrix shows the causal relations ($\rightarrow$) between

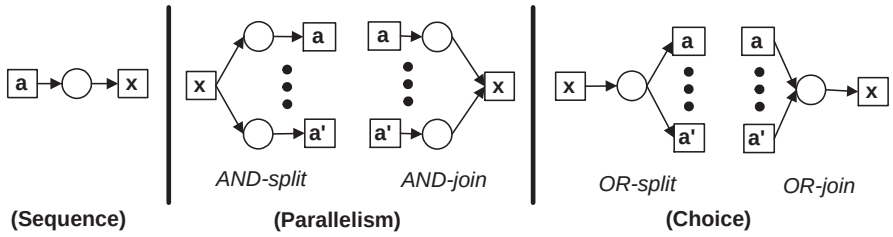


Fig. 2. Petri net building blocks for the three basic routing constructs that are used when modelling business processes.

the activities in the process. For this reason, we call it the *causal matrix*. The causal matrix has size $n \times n$, where n is the number of process' activities. The boolean expressions are used to describe the routing constructs. Because the boolean expressions describe AND/OR-split/join situations, they only contain the boolean operators *and* ($\wedge$) and *or* ($\vee$).

As an example, we show how the Petri net in Figure 1 can be described by the casual matrix shown in Table 2. The Petri net in Figure 1 has 8 activities ($A...H$), so the corresponding individual is represented by an 8×8 causal matrix. An entry (row, column) in the causal matrix describes if there is a causal relation between two activities. If *causal*(row, column) = 1, there is such a causal relation. If it equals 0, there is no such relation. The boolean expressions in the INPUT row describe which activities should occur to enable the occurrence of an activity at a column. For instance, consider activity H in Figure 1. This activity can occur whenever activity B *or* C *or* G occurs. Thus, column H has the boolean expression $B \vee C \vee G$ associated to it. Similarly, the boolean expressions in the OUTPUT column show which activities may execute after the execution of an activity at a row. For instance, row D has as OUTPUT the boolean expression $E \wedge F$.

	INPUT								
	true	A	A	A	D	D	$E \wedge F$	$B \vee C \vee G$	
$\rightarrow$	A	B	C	D	E	F	G	H	*OUTPUT*
A	0	1	1	1	0	0	0	0	$B \vee C \vee D$
B	0	0	0	0	0	0	0	1	H
C	0	0	0	0	0	0	0	1	H
D	0	0	0	0	1	1	0	0	$E \wedge F$
E	0	0	0	0	0	0	1	0	G
F	0	0	0	0	0	0	1	0	G
G	0	0	0	0	0	0	0	1	H
H	0	0	0	0	0	0	0	0	*true*

Table 2. A causal matrix is used for the internal representation of an individual.

ACTIVITY	*INPUT*	*OUTPUT*
A	$\{\}$	$\{\{B, C, D\}\}$
B	$\{\{A\}\}$	$\{\{H\}\}$
C	$\{\{A\}\}$	$\{\{H\}\}$
D	$\{\{A\}\}$	$\{\{E\}, \{F\}\}$
E	$\{\{D\}\}$	$\{\{G\}\}$
F	$\{\{D\}\}$	$\{\{G\}\}$
G	$\{\{E\}, \{F\}\}$	$\{\{H\}\}$
H	$\{\{B, C, G\}\}$	$\{\}$

Table 3. A more succinct encoding of the individual shown in Table 2.

Given the conceptual description of individuals, let us explain how it is actually encoded in our genetic algorithm[1]. First of all, the algorithm only keeps track of an individual's activities' INPUT and OUTPUT boolean expressions. Because the complete causal matrix can be directly derived from the boolean expressions, the causal matrix is not explicitly stored but only used during the initialization process. By looking at the boolean expressions you derive which entries are set to 1 and which are set to 0 in the causal matrix. Second, the boolean expressions are mapped to sets of subsets. Activities in a subset have an OR-relation and subsets are in an AND-relation. For instance, the boolean expression $(E \vee F) \wedge G$ equals the set representation $\{\{E, F\}, \{G\}\}$. Table 3 shows how the conceptual encoding in Table 2 is mapped to the implementation one. Note that Table 3 assumes a "normal form", i.e., a conjunction of disjunctions. This reduces the state space but also limits the expressiveness, cf. Appendix A.

2.2 Parsing Semantics

Our GA mining approach searches for a process model that is in accordance with the information in the event log. Testing if all traces can be parsed by the mined process model is one possibility to check this. The parsing semantics of a process model is relatively simple. It sequentially reads one activity at a time from an event trace and it checks if this activity can be executed or not. An activity can execute when its INPUT boolean expression is true (i.e. at least one of the activities of each subset has the value 1). Let us use an example to clarify how the parsing works. Consider the parsing of the event trace for *case 3* in Table 1 - the trace "A, D, E, F, G, H" - and the process model described in Table 2. The parsing of this trace is depicted in Figure 3. The element being parsed (see left column) is in gray. The right column shows which activities' markings of the individual have being affected by the previous parsed element (also highlighted in gray). Note that parsing an element affects the marking of the activities in its OUTPUT boolean expression. The values (0 or bigger) are used to keep track of true (= 1) or false (= 0) value of the individual marking elements. Besides, because start activities have a single input place and end activities have a single output place, we use two auxiliary elements in the marking: *start* and *end*. Row

[1] A detailed explanation of the genetic algorithm is given in Section 3

(i) shows the initial situation. A is the first activity to be parsed. Its INPUT boolean expression is true. This means that activity A is a start activity and A can be executed whenever the start element has the value 1. This is indeed the situation at row (i). After executing A, the activities's markings are updated. In this case, the *start* element gets value 0 and the activities associated to A's OUTPUT get their values increased by 1. Note that during the marking update, OR-situations are treated in a different way of AND-situations. As an example of an OR-situation, consider row (ii) in which D is the activity to be parsed. D can be executed because its INPUT shows that it can be executed whenever A has the entry $D = 1$ in its marking. However, the execution of D also affects A's marking for activities B and C because A's OUTPUT describes that activities B, C and D have an OR-relation. The final result is at row (iii), which contains an example of an AND-situation. At row (iii), E is the next activity to be parsed. Note that E can be parsed, but the related activities' markings are updated in a different way from the situation just described for the parsing of D. The entry $D : ..., F = 1$ is not affected because D's OUTPUT shows that E and F are in an AND-situation. Thus, the execution of E does not disable the execution of F, and vice-versa. As shown in Figure 3, the trace "A, D, E, F, G, H" is indeed successfully parsed by the individual in Table 2 because the *end* element is the only one to be marked 1 when the parsing stops.

3 Genetic Algorithm

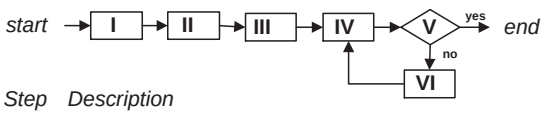
In this section we explain how our genetic algorithm works. Figure 4 describes its main steps. The following subsections respectively describe (i) the initialization process, (ii) the fitness calculation, (iii) the stop criteria and (iv) the genetic operators (i.e. crossover and mutation) of our genetic algorithm.

3.1 Initialization of the population

If we directly use the INPUT and OUTPUT-subsets for the initialization of our start population, the number of different possible individuals appears enormous. If n is the number of activities in the event log, the number of different process models is roughly $(2^n \times 2^n)^n$. Even for the simple example this results in 2^{128} different possibilities. Therefore we chose to guide the genetic algorithm during the building of the initial population by using a *dependency measure*. The measurements are based on our experience with the heuristic mining tool *Little's Thumb* [42]. In the next paragraph we explain how the dependency measure is used during initialization. The main idea is that if the substring "t_1t_2" appears frequently and "t_2t_1" only as an exception, than there is a high probability that t_1 and t_2 are in a causal relation. We use $follows(t_1, t_2)$ as an notation for the number of times that the substring t_1t_2 appears in the event log, and $causal(t_1, t_2) = 1$ to indicate that the causal matrix has the value 1 in row t_1 and column t_2. Before we present our definition of the dependency measure we need two extra notations for short loops: $L1L(t_1)$ indicates the number of times the substring "t_1t_1"

	Element being parsed	Individual's current marking	
(i)	A, D, E, F, G, H	A: B = 0, C = 0, D = 0 B: H = 0 C: H = 0 D: E = 0, F = 0 E: G = 0	F: G = 0 G: H = 0 start = 1 end = 0
(ii)	A, D, E, F, G, H	A: B = 1, C = 1, D = 1 B: H = 0 C: H = 0 D: E = 0, F = 0 E: G = 0	F: G = 0 G: H = 0 start = 0 end = 0
(iii)	A, D, E, F, G, H	A: B = 0, C = 0, D = 0 B: H = 0 C: H = 0 D: E = 1, F = 1 E: G = 0	F: G = 0 G: H = 0 start = 0 end = 0
(iv)	A, D, E, F, G, H	A: B = 0, C = 0, D = 0 B: H = 0 C: H = 0 D: E = 0, F = 1 E: G = 1	F: G = 0 G: H = 0 start = 0 end = 0
(v)	A, D, E, F, G, H	A: B = 0, C = 0, D = 0 B: H = 0 C: H = 0 D: E = 0, F = 0 E: G = 1	F: G = 1 G: H = 0 start = 0 end = 0
(vi)	A, D, E, F, G, H	A: B = 0, C = 0, D = 0 B: H = 0 C: H = 0 D: E = 0, F = 0 E: G = 0	F: G = 0 G: H = 1 start = 0 end = 0
(vii)	A, D, E, F, G, H	A: B = 0, C = 0, D = 0 B: H = 0 C: H = 0 D: E = 0, F = 0 E: G = 0	F: G = 0 G: H = 0 start = 0 end = 1

Fig. 3. Illustration of the parsing process of the event trace A, D, E, F, G, H for case 3 in Table 1 by the process model in Table 2.



Step	Description
I	Read event log
II	Calculate dependency relations among activities
III	Build the initial population
IV	Calculate individuals' fitness
V	Stop and return the fittest individuals?
VI	Create next population - use genetic operations

Fig. 4. Main steps of our genetic algorithm.

appears in the event log (length-one loop) and $L2L(t_1, t_2)$ the number of times the substring "$t_1t_2t_1$" appears (length-two loop). The dependency measure is defined as follows:

Definition 3.1. (Dependency Measure) Let t_1 and t_2 be two activities in event log T. Then:

$$D(t_1,t_2) = \begin{cases} \frac{L2L(t_1,t_2) + L2L(t_2,t_1)}{L2L(t_1,t_2) + L2L(t_2,t_1) + 1} & \text{if } t_1 \neq t_2 \text{ and } L2L(t_1,t_2) > 0 \\ \frac{follows(t_1,t_2) - follows(t_2,t_1)}{follows(t_1,t_2) + follows(t_2,t_1) + 1} & \text{if } t_1 \neq t_2 \text{ and } L2L(t_1,t_2) = 0 \\ \frac{L1L(t_1,t_2)}{L1L(t_1,t_2) + 1} & \text{if } t_1 = t_2 \end{cases}$$

The "+ 1" in the denominator of Definition 3.1 is used to benefit more frequent occurrences. Additionally to the dependency measure, we use a *start*-measure and an *end*-measure. These two measures are used to determine the *start* and *end* activity of a mined process model. To calculate them we simple add an additional activity *start* and *end* to each trace. The start measure for activity t (notation $S(t)$) is equal to $D(start, t)$ and the end measure (notation $E(t)$) to $E(t, end)$.

Building the initial population is a random process driven by the dependency measures between activities. First we determine the boolean values of the causal matrix. The basic idea is that if, for two activities t_1, t_2, the dependency measure $D(t_1, t_2)$ is high than there is a high probability that $causal(t_1, t_2)$ is true (value 1). Below the procedure for the initialization of a process model is given.

1. For all activities t_1 and t_2 generate a random number r. If $r < (D(t_1, t_2))^p$ then $causal(t_1, t_2) = 1$ else $causal(t_1, t_2) = 0$.
2. For all activities t if $r < (S(t))^p$ then the complete $causal(t)$-column is set to 0.
3. For all activities t if $r < (E(t))^p$ then the complete $causal(t)$-row is set to 0.
4. For every column t_1 in the causal matrix the INPUT set is a random partition of the set $X_i := \{t_2 | causal(t_2, t_1) = 1\}$.
5. For every row t_1 in the causal matrix the OUTPUT set is a random partition of the set $X_i := \{t_2 | causal(t_1, t_2) = 1\}$.

The power value p is introduced to manipulate the eagerness of the initialization process to introduce causal relations. Note that p needs to be odd to keep negative values negative. A high value of p (e.g., $p = 9$) results in relatively few causal relations, a low value in relatively many causal relations (e.g., $p = 1$). For every entry in the causal matrix a new random number r is drawn. Activities with a high S-value (*start*-value) have a high probability that the complete column is set to 0 and activities with a high E-value have a high probability that the complete row is 0. This is done because, as explained in Section 2, the algorithm assumes that *start*-activities have a single input place (which does not have ingoing arcs), and *end*-activities have a single output place (which does not have outgoing arcs). For every column in the causal matrix, the algorithm retrieves the activities whose entry (activity, column) equals 1. These activities are randomly combined in a boolean INPUT expression that (i) does not repeat symbols (i.e. an activity cannot appear more than once in a boolean expression) and (ii) is a conjunction of disjuncts. As an example, consider activity H in the causal matrix in Table 2. The retrieved activities for column H are B, C and G. So, the possible random combinations for these three activities are: $B \wedge C \wedge G$, $(B \wedge C) \vee G$, $(B \wedge G) \vee C$, $(C \wedge G) \vee B$, $B \vee G \vee C$. The analogue procedure is used to construct an OUTPUT expression.

Individual1

ACTIVITY	INPUT	OUTPUT
A	$\{\}$	$\{\{B, C, D\}\}$
B	$\{\{A\}\}$	$\{\{H\}\}$
C	$\{\{A\}\}$	$\{\{H\}\}$
D	$\{\{A\}\}$	$\{\{E\}\}$
E	$\{\{D\}\}$	$\{\{G\}\}$
F	$\{\}$	$\{\{G\}\}$
G	$\{\{E\}, \{F\}\}$	$\{\{H\}\}$
H	$\{\{C, B, G\}\}$	$\{\}$

Individual2

ACTIVITY	INPUT	OUTPUT
A	$\{\}$	$\{\{B, C, D\}\}$
B	$\{\{A\}\}$	$\{\{H\}\}$
C	$\{\{A\}\}$	$\{\{H\}\}$
D	$\{\{A\}\}$	$\{\{E, F\}\}$
E	$\{\{D\}\}$	$\{\{G\}\}$
F	$\{\{D\}\}$	$\{\{G\}\}$
G	$\{\{E\}, \{F\}\}$	$\{\{H\}\}$
H	$\{\{C\}, \{B\}, \{G\}\}$	$\{\}$

Table 4. Causal matrix of two randomly created individuals for the log in Table 1.

As an example, we show in Table 4 and in Figure 5 two individuals that could be randomly built for the initial population, for the log in Table 1. The next step in the genetic algorithm is the calculation of the fitness of individuals.

3.2 Fitness Calculation

For a noise-free log, the genetic search aims at finding an optimal process that complies with the information in the event log. Testing if all traces can be parsed by the mined process model PM is one possibility to check this[2]. Thus, a simple

[2] Normally, we don't have negative examples at our disposal. If we have negative examples, we can check if the parsing indeed fails.

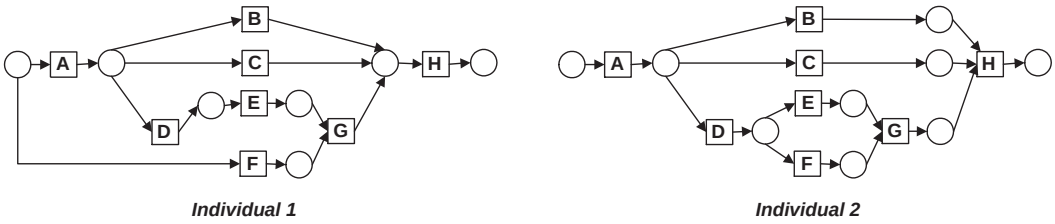


Fig. 5. Petri net of two randomly created individuals for the log in Table 1.

fitness measure can just calculate the number of correct parsed traces divided by the number of traces in the log L. However, such a fitness measure is too naive because it gives a very coarse indication about a process model's compliance to a given log. For instance, assume that for one process model PM_1 the parsing usually gets stuck in the first part of a trace and in an other process model PM_2 usually at the end of the trace. Although PM_2 is a better candidate to crossover because it contains more correct material, this fitness does not indicate that. Moreover, we like a proper completion of the parsing process. This means that only the value of the auxiliary element *end* equals 1 (cf. Subsection 2.2) and all the other values are 0. In Definition 3.2 we present a fitness measure that incorporate these observations. The notation used is as follows. *numActivitiesLog*(L) and *numTracesLog*(L) respectively indicate the number of activities and traces in the log. For instance, the log in Table 1 has 18 activities and 4 traces. *allParsedActivities*(PM, L) gives the sum of all parsed activities for all traces in the event log. *allCompletedLogTraces*(PM, L) gives the number of completely parsed traces. *allProperlyCompletedLogTraces*(PM, L) gives the number of completely parsed traces in which the auxiliary element *end* equals 1 and all other values equals 0. Note the subscript "$_S$" for some of the terms in Definition 3.2. These are used to distinguish the "Stop semantics" from the "Continuous semantics" (we will elaborate on this later). Also note the three coefficients in this definition. We did some experiments to get reasonable coefficient values for both fitness measures presented in this section.

Definition 3.2. (Fitness$_S$) Let L be an event log and PM be a process model. Then:

$$Fitness_S(PM, L) = 0.20 \times \frac{allParsedActivities_S(PM,L)}{numActivitiesLog(L)} +$$

$$0.30 \times \frac{allCompletedLogTraces_S(PM,L)}{numTracesLog(L)} + 0.50 \times \frac{allProperlyCompletedLogTraces_S(PM,L)}{numTracesLog(L)}$$

Definition 3.2 assumes a stop semantics, i.e., when parsing event traces the parsing stops the moment the log indicates that an activity should be executed while this is not possible in the process model. All remaining events in the event trace are subsequently ignored. As a result, $Fitness_S$ has the disadvantage that it will stop parsing whenever a parsing error occurs (the subscript S in the naming of $Fitness_S$ indicates the stop semantics). A consequence is that if we have two process models PM_1 and PM_2, where PM_1 has only one error close to an start

activity and PM_2 has the same error but also many errors the remainder of its net structure, the fitness of both models will be equal. Also errors that occur at the start of a model have a higher penalty than errors at the end of the model. Repairing this problem is obvious: simply do not stop the parsing process after identifying an error. Instead, register the error and go on with the parsing process. Another possible gain of this continuous semantics parsing procedure is a better behavior in case of noisy traces because it gives information about the complete process model (i.e. not biased to only the first part of the process model) and the behavior for the whole trace (not only for the first, error free part of a trace). The fitness measure in Definition 3.3 incorporates such a continuous semantics. (Note the subscript "$_C$".)

Definition 3.3. (Fitness$_C$) Let L be an event log and PM be a process model. Then:

$$Fitness_C(PM, L) =$$

$$0.40 \times \frac{allParsedActivities_C(PM,L)}{numActivitiesLog(L)} + 0.60 \times \frac{allProperlyCompletedLogTraces_C(PM,L)}{numTracesLog(L)}$$

In the next section we will report our experimental results for both fitness measures and their behavior in case of noise in the event log. But first we will finish this section with describing more details of our GA.

3.3 Stop Criteria

The mining algorithm stops when (i) it finds an individual with a fitness of 1; or (ii) it computes n generations, where n is the maximum number of generation that is allowed; or (iii) the fittest individual has not changed for $n/2$ generations in a row. When the algorithm does not stop, it creates a new population by using the genetic operations that are described in the next section.

3.4 Genetic Operations

We use elitism, crossover and mutation to build the individuals of the next genetic generation. Elitism means that a percentage of the fittest individuals in the current generation is copied to the next generation. Crossover and mutation are the basic genetic operations. Crossover creates new individuals (offsprings) based on the fittest individuals (parents) in the current population. So, crossover recombines the fittest material in the current population in the hope that the recombination of useful material in one of the parents will generate an even fitter individual. The mutation operation will change some minor details of an individual. The hope is that the mutation operator will insert new useful material in the population. In this section we show the crossover and mutation algorithms that turned out to give good results during our experiments. The algorithm to create a next generation works as follows:

***Input**: current population, elitism rate, crossover rate and mutation rate*
***Output**: new population*

1. *Copy "elitism rate × population size" of the best individuals in the current population to the next population.*
2. *While there are individuals to be created do:*
   (a) *Use tournament selection to select* $parent_1$.
   (b) *Use tournament selection to select* $parent_2$.
   (c) *Select a random number* r *between* 0 *(inclusive) and* 1 *(exclusive).*
   (d) *If* r *less than the crossover rate:*
   then *do crossover with* $parent_1$ *and* $parent_2$. *This operation generates two offsprings:* $offspring_1$ *and* $offspring_2$.
   else $offspring_1$ *equals* $parent_1$ *and* $offspring_2$ *equals* $parent_2$.
   (e) *Mutate* $offspring_1$ *and* $offspring_2$. *(This step is only needed if the mutation rate is non-zero.)*
   (f) *Copy* $offspring_1$ *and* $offspring_2$ *to the new population.*
3. *Return the new population.*

Tournament Selection The tournament selection is used to select two parents to crossover. Given a population, it randomly selects 5 individuals and it returns the fittest individual among the five selected ones.

Crossover An important operation in our genetic approach is the crossover operation. This is also the most complex genetic operation. Starting point of the crossover operation are two parents (i.e. $parent_1$ and $parent_2$). The result of applying the crossover operation are two offsprings ($offspring_1$ and $offspring_2$). First, the crossover algorithm randomly selects an activity t to be the *crossover point*. Second, $parent_1$ is copied to $offspring_1$ and $parent_2$ to $offspring_2$. Third, the algorithm randomly selects a *swap point* for the INPUT(t) sets in both offsprings and another *swap point* for the OUTPUT(t) sets. The respective INPUT and OUTPUT sets of the crossover point at the two offsprings are then recombined by interchanging the subsets from the swap point until the end of the set. The recombined INPUT/OUTPUT sets are then checked to make sure that they are proper partitions. Finally, the two offsprings undergo a repair operation called "update related elements". The pseudo-code for the crossover is as follows:

Input*: Two individuals*
Output*: Two recombined individuals*

1. *If the individuals are equal, go to Step 11.*
2. *Randomly select an activity* t *to be the individuals' crossover point.*
3. *Set1 = INPUT(t) in the first individual.*
4. *Set2 = INPUT(t) in the second individual.*
5. *Select a swap point* $sp1$ *to crossover in Set1. The swap point has a value between 0 (before the first subset) and the number of subsets in the set minus 1.*
6. *Select a swap point* $sp2$ *to crossover in Set2.*
7. *Swap the selected parts. The parts go from the swap point to the end of the set.*

8. *If there are overlaps in the subsets, with an equal probability* either *merge the sets whose intersection is non-empty* or *remove the intersecting activities from the subset that is not being swapped.*
9. *Update the related activities.*
10. *Repeat steps 3 to 9 but use the OUTPUT sets instead of the INPUT sets.*
11. *Return the two recombined individuals.*

Update Related Activities When the individuals have different causal matrices, the crossover operation may generate inconsistencies. Note that the boolean expression may contain activities whose respective cell in the causal matrix is zero. Similarly, an activity may not appear in the boolean expression after the crossover and the causal matrix still has a non-zero entry for it. So, after the INPUT/OUTPUT sets have being recombined, we need to check the consistency of the recombined sets with respect to the other activities' boolean expressions and the causal matrix. When they are inconsistent, we need to update the causal matrix and the related boolean expressions of the other activities. The algorithm works as follows:

Input*: an individual, an activity t that was the crossover point*
Output*: an updated individual*

1. *Update the causal matrix.*
   Explanation: The INPUT(t) is used to update the column t in the causal matrix. The OUTPUT(t) is used to update the row t in the causal matrix. Every activity t' in the INPUT(t) has $causal(t', t) = 1$. All the other entries at the column are set to zero. A similar procedure is done for the activities in OUTPUT(t).
2. *Check the boolean expressions of the other activities against the column and row for t in the causal matrix.*
   Explanation: Whenever there are inconsistencies between the entries in the causal matrix and the boolean expression, the activities whose entry is zero in the causal matrix are eliminated from the respective boolean expression, and activities whose entry is 1 are included in one of the subsets in the boolean expression.

Figure 6 illustrates a crossover operation that involves the two individuals in Figure 5. Let activity D be the randomly selected crossover point. Since INPUT1(D) equals INPUT2(D), the crossover has no real effect for D's INPUT. Let us look at the D's OUTPUT sets. Both D's OUTPUT sets have a single subset, so the only possible swap point to select equals 0, i.e., before the first and only element. After swapping the subsets *Offpring1* ($parent_1$ after crossover) has INPUT1(D)= $\{\{A\}\}$ and OUTPUT1(D)= $\{\{E, F\}\}$. Note that OUTPUT1(D) now also points to F. So, the *update related elements* algorithm makes INPUT1(F)= $\{\{D\}\}$. $offspring_2$ is updated in a similar way. The internal representation for the two offsprings is showns in Table 5.

Mutation The mutation works on the INPUT and OUTPUT boolean expressions of an activity. For every activity t in an individual, a new random number

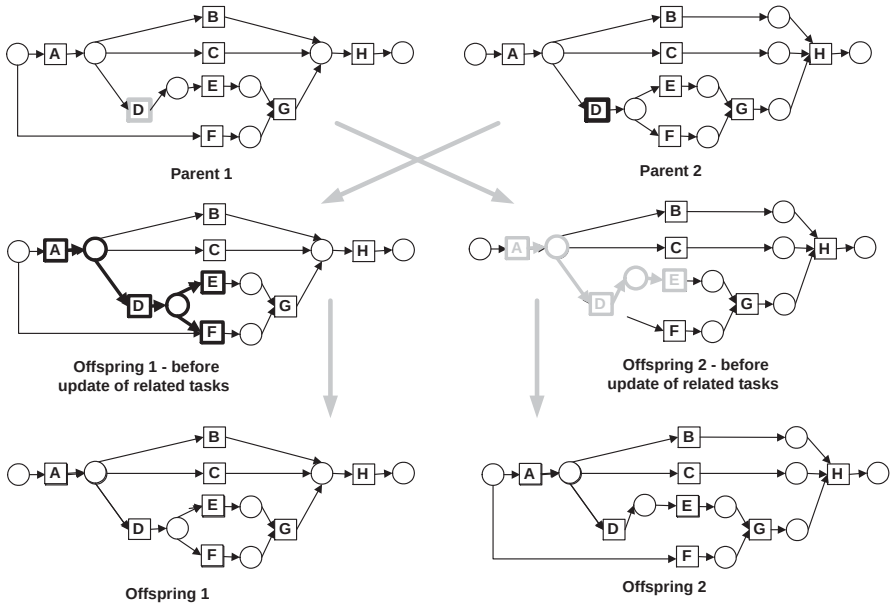


Fig. 6. Example of the crossover operation for the two individuals in Figure 5. The crossover point is activity D.

$offspring_1$		
ACTIVITY	**INPUT**	**OUTPUT**
A	$\{\}$	$\{\{B, C, D\}\}$
B	$\{\{A\}\}$	$\{\{H\}\}$
C	$\{\{A\}\}$	$\{\{H\}\}$
D	$\{\{A\}\}$	$\{\{E, F\}\}$
E	$\{\{D\}\}$	$\{\{G\}\}$
F	$\{\{D\}\}$	$\{\{G\}\}$
G	$\{\{E\}, \{F\}\}$	$\{\{H\}\}$
H	$\{\{C, B, G\}\}$	$\{\}$

$offspring_2$		
ACTIVITY	**INPUT**	**OUTPUT**
A	$\{\}$	$\{\{B, C, D\}\}$
B	$\{\{A\}\}$	$\{\{H\}\}$
C	$\{\{A\}\}$	$\{\{H\}\}$
D	$\{\{A\}\}$	$\{\{E\}\}$
E	$\{\{D\}\}$	$\{\{G\}\}$
F	$\{\}$	$\{\{G\}\}$
G	$\{\{E\}, \{F\}\}$	$\{\{H\}\}$
H	$\{\{C\}, \{B\}, \{G\}\}$	$\{\}$

Table 5. Example of two offsprings that can be produced after a crossover between the two individuals in Table 4. The crossover point is activity D.

r is selected. Whenever r less than the "mutation rate", the subsets in INPUT(t) are randomly merged or split. The same happens to OUTPUT(t). The mutation algorithm works as follows:

Input: *an individual*
Output: *a possibly mutated individual.*

1. *For every activity in the individual do:*
   (a) *Select a random number r between* 0 *(inclusive) and* 1 *(exclusive).*
   (b) *If r less than the specified mutation rate:*
      i. *Build a new expression for the INPUT of this activity.*
      ii. *Build a new expression for the OUTPUT of this activity.*
2. *Return the individual.*

As an example, consider $offspring_1$ in Table 5. Assume that the random number r was less than the mutation rate for activity D. After applying the mutation, OUTPUT(D) changes from $\{\{E, F\}\}$ to $\{\{E\}, \{F\}\}$. Note that this mutation does not change an individual's causal relations, only its AND-OR/join-split may change.

4 Experiments and Results

To test our genetic approach and the effect of the two different fitness measures $Fitness_S$ and $Fitness_C$ we use 4 different process models with 8, 12, 22 and 32 activities. These nets are respectively described in Figures 1, 7, 8 and 9. The nets were artificially generated and contain concurrency and loops. To test the behavior of the genetic algorithm for event logs with noise, we used 6 different noise types: *missing head*, *missing body*, *missing tail*, *missing activity*, *exchanged activities* and *mixed noise*. If we assume a event trace $\sigma = t_1...t_{n-1}t_n$, these noise types behave as follows. *Missing head, body* and *tail* respectively randomly remove subtraces of activities in the head, body and tail of σ. The head goes from t_1 to $t_{n/3}$. The body goes from $t_{(n/3)+1}$ to $t_{(2n/3)}$. The tail goes from $t_{(2n/3)+1}$ to t_n. *Missing activity* randomly removes *one* activity from σ. *Exchanged activities* exchange two activities in σ. *Mixed noise* is a fair mix of the other 5 noise types. Real life logs will typically contain mixed noise. However, the separation between the noise types allow us to better assess how the different noise types affect the genetic algorithm.

For every noise type, we generated logs with 5%, 10% and 20% of noise. So, every process model in our experiments had $6 \times 3 = 18$ noisy logs. Besides, we also ran the experiments for noisy-free logs of the process models because our approach should also work for noisy-free logs. Thus, every process model in our experiments has in total 19 logs. Every event log had 1000 traces. For each event-log the genetic algorithms ran 10 experiments with different seeds. The populations had 500 individuals and were iterated for at most 100 generations. The crossover rate was 1.0 and the mutation rate was 0.01. The elitism rate was

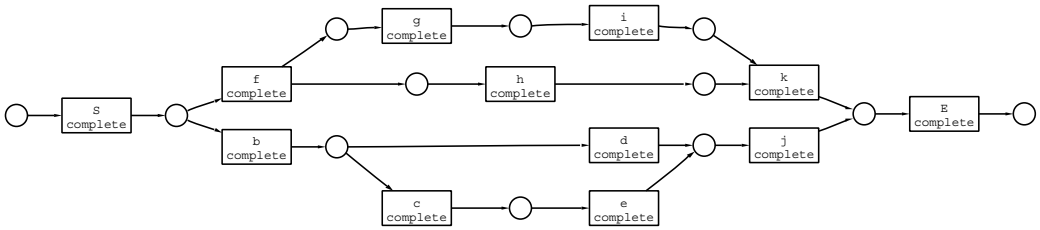


Fig. 7. Petri net for process model with 12 activities.

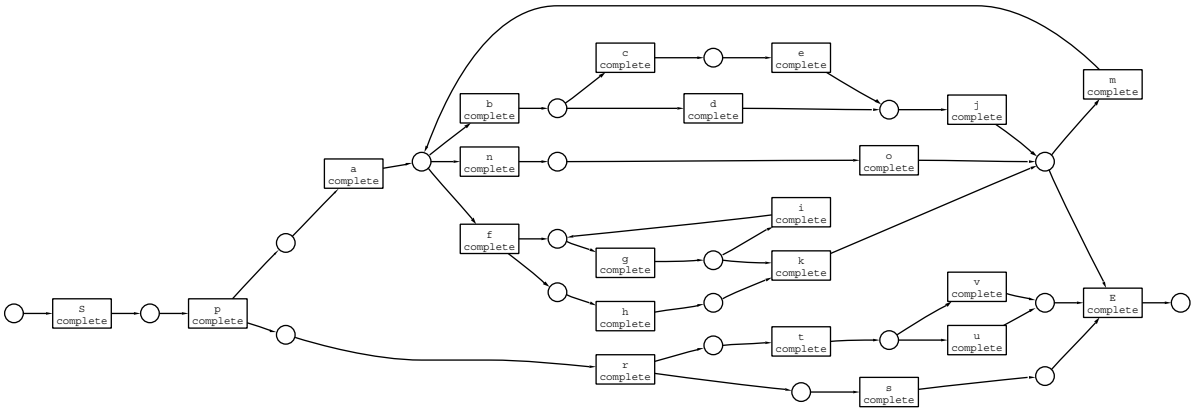


Fig. 8. Petri net for process model with 22 activities.

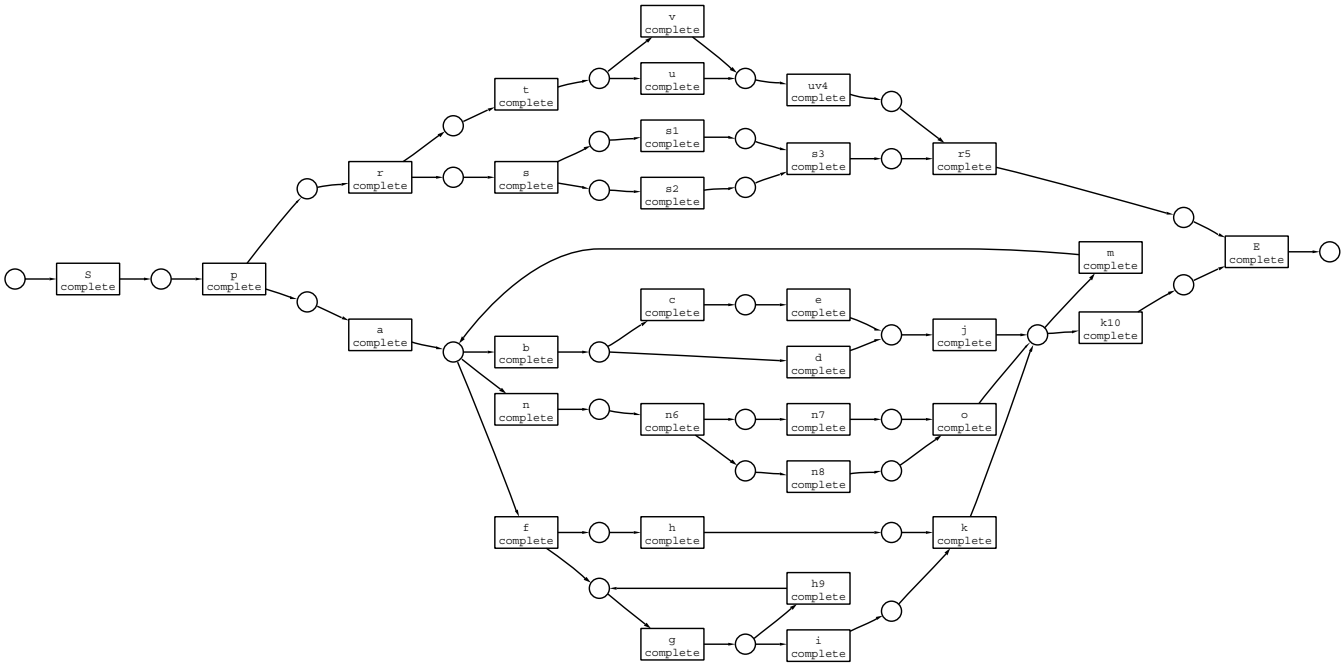


Fig. 9. Petri net for process model with 32 activities.

0.01. The power for the causal relation (cf. Subsection 3.1) was 9. The initial population might contain duplicate individuals.

An important general question is how to measure the quality of mined models in the case of noisy logs. In the experimental setting we know that the model that is used to generate the event logs and you may expect that the genetic algorithm will come up with exactly this model. In a more realistic situation, you will not know the underlying model, you are searching for it. The problem is that it is very difficult to distinguish low frequent behavior from noise. Not modelled low frequent possible behavior registered in the event log can be interpreted as an error. However, low or even high frequent registered noise that is incorporated in the model are errors. In a practical situation the only sensible solution seems the definition of an appropriate fitness measure. However, in our experimental setting we are experimenting with different fitness measures. Therefore we cannot use one of them as *the* measure. In our experimental setting the simplest solution to measure the quality of an genetic algorithm is counting the number of runs in which the genetic search comes up with exactly the process model that is used during the creation of the noise-free event logs. Even in the case that noise is added to the event log we will use this measure.

Let us first have a look at the results for the noisy-free logs. As shown in Figure 10 and Table 6, the genetic algorithm works for noise-free logs. For both fitness types, the smaller the net, the more frequently the algorithm finds the

desired process model. For process models that contain more activities, the GA using $Fitness_C$ seems to work better than the GA using $Fitness_S$. Although the correct process model was not found for all runs (25 out of 40 for the GA using $Fitness_S$ and 32 out of 40 for the GA using $Fitness_C$), the other runs returned nearly correct individuals.

However, our main aim is to use genetic algorithms to mine noisy logs. The results for the *mixed noise* type in Figures 11 to 13 show that the genetic algorithm indeed works for noisy logs as well. Again we see that the smaller the net, the more frequently the algorithm finds the correct process model; and the higher the noise percentage, the lower the probability the algorithm will end up with the original process model. However, the GA using $Fitness_C$ is more robust to noise. Tables 7 to 10 have the detailed results. By looking at the results for the different noise types we have the following observations. The algorithm can handle well the *missing tail* noise type for both fitness types because of the high impact of *proper completion*. The *Missing head* impacts more the experiments using the $Fitness_C$ than the $Fitness_S$ because the former fitness punishes more the process models that do not properly complete. The *exchanged activities* noise type impacts less the performance of the algorithm than the *missing body* and the *missing activity* noise types because of the heuristics that are used during the building of the initial population. Removing an activity t_2 from a trace "$...t_1t_2t_3...$" generates "fake" subtraces t_1t_3 that will not be counter balanced by subtraces t_3t_1. Consequently, the probability that the algorithm will causally relate t_1 and t_3 is increased.

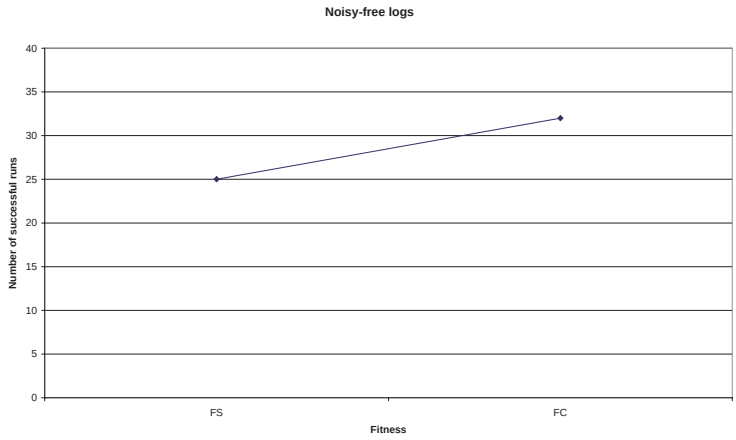


Fig. 10. Results for noisy-free logs. *FS* and *FC* respectively show the results for $Fitness_S$ and $Fitness_C$.

In this section we presented some results for the genetic mining approach presented in this paper. In contrast to most of the existing approaches, our GA process mining is able to deal with noise. However, more improvements are

number of activities in process model	*number of successful runs*	
	FS	*FC*
8	10	10
12	10	10
22	02	04
32	03	08

Table 6. Results of applying the genetic algorithm for noise-free logs. The table shows the number of times the perfect individual was found in 10 runs. *FS* and *FC* respectively show the results for $Fitness_S$ and $Fitness_C$.

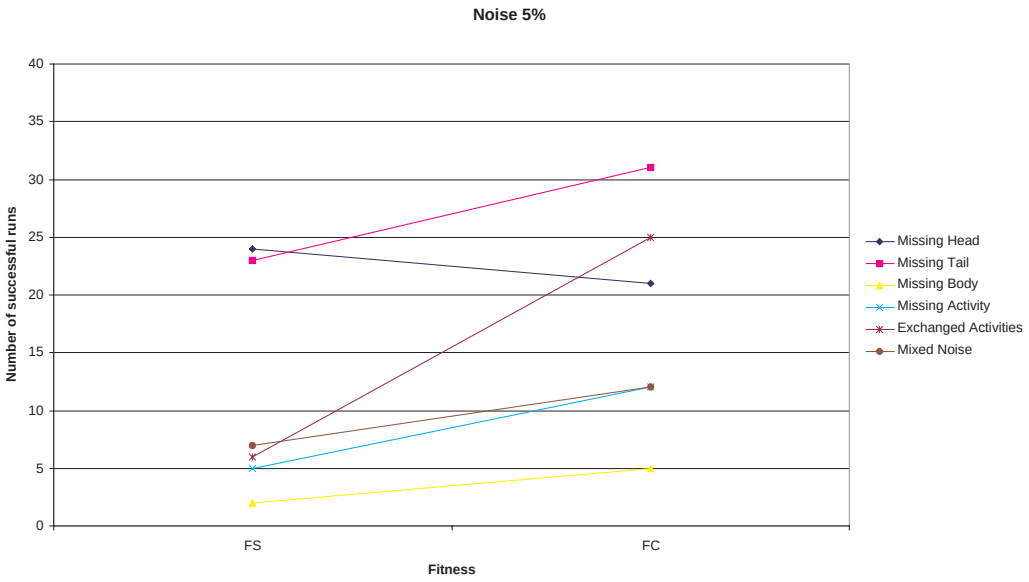


Fig. 11. Results for logs with 5% of noise. *FS* and *FC* respectively show the results for $Fitness_S$ and $Fitness_C$.

	noise type											
noise	*Missing head*		*Missing tail*		*Missing body*		*Missing activity*		*Exchanged activities*		*Mixed noise*	
percentage	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*
5%	10	10	10	10	0	0	5	1	3	9	3	1
10%	10	10	10	10	0	1	1	1	3	5	1	3
20%	10	0	10	10	0	0	0	0	0	0	1	2

Table 7. Results of applying the genetic algorithm for noisy logs of the process models with *8 activities*. The table shows the number of times the perfect individual was found in *10 runs*. *FS* and *FC* respectively show the results for $Fitness_S$ and $Fitness_C$.

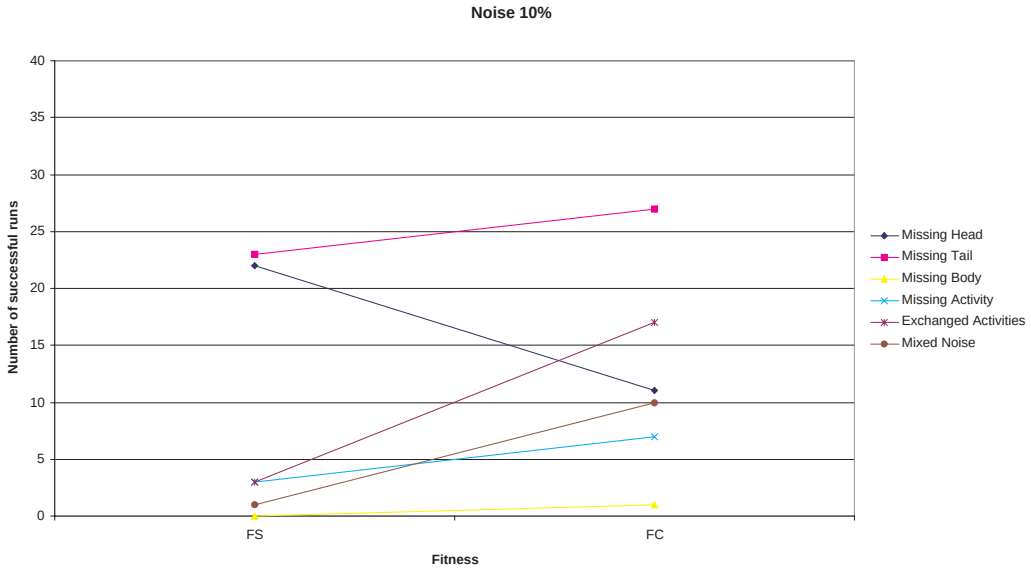


Fig. 12. Results for logs with 10% of noise. *FS* and *FC* respectively show the results for $Fitness_S$ and $Fitness_C$.

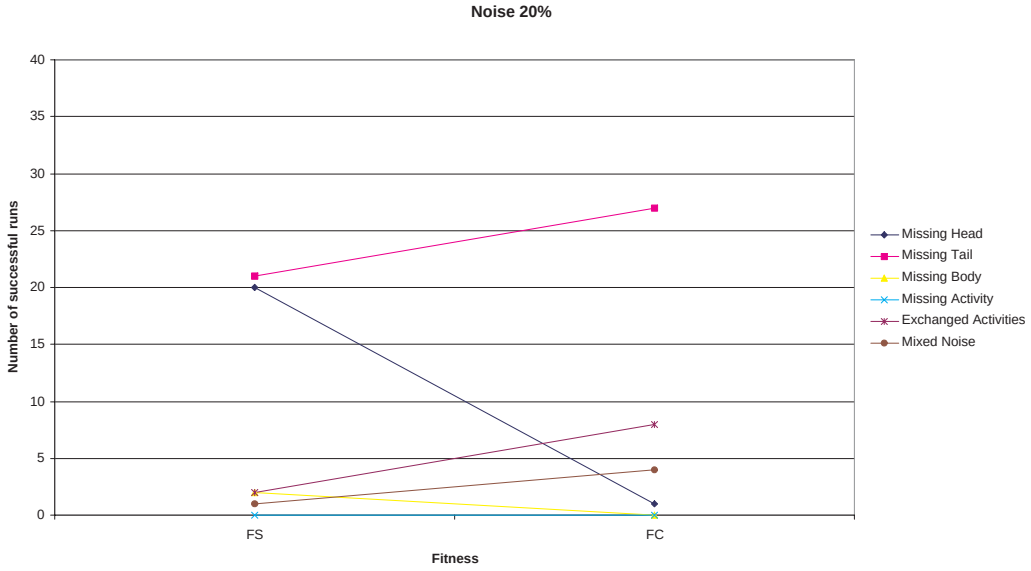


Fig. 13. Results for logs with 20% of noise. *FS* and *FC* respectively show the results for $Fitness_S$ and $Fitness_C$.

	noise type											
noise	*Missing head*		*Missing tail*		*Missing body*		*Missing activity*		*Exchanged activities*		*Mixed noise*	
percentage	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*
5%	10	10	10	10	0	0	0	0	2	10	3	2
10%	10	1	10	10	0	0	0	0	0	9	0	3
20%	10	1	10	10	0	0	0	0	2	8	0	2

Table 8. Results of applying the genetic algorithm for noisy logs of the process models with *12 activities*. The table shows the number of times the perfect individual was found in *10 runs*. *FS* and *FC* respectively show the results for $Fitness_S$ and $Fitness_C$.

	noise type											
noise	*Missing head*		*Missing tail*		*Missing body*		*Missing activity*		*Exchanged activities*		*Mixed noise*	
percentage	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*
5%	2	1	0	5	0	2	0	6	0	4	0	4
10%	0	0	0	2	0	0	0	1	0	3	0	0
20%	0	0	0	5	0	0	0	0	0	0	0	0

Table 9. Results of applying the genetic algorithm for noisy logs of the process models with *22 activities*. The table shows the number of times the perfect individual was found in *10 runs*. *FS* and *FC* respectively show the results for $Fitness_S$ and $Fitness_C$.

	noise type											
noise	*Missing head*		*Missing tail*		*Missing body*		*Missing activity*		*Exchanged activities*		*Mixed noise*	
percentage	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*	*FS*	*FC*
5%	2	0	3	6	2	3	0	5	1	2	1	6
10%	2	0	3	5	0	0	2	5	0	0	0	4
20%	0	0	1	2	2	0	0	0	0	0	0	0

Table 10. Results of applying the genetic algorithm for noisy logs of the process models with *32 activities*. The table shows the number of times the perfect individual was found in *10 runs*. *FS* and *FC* respectively show the results for $Fitness_S$ and $Fitness_C$.

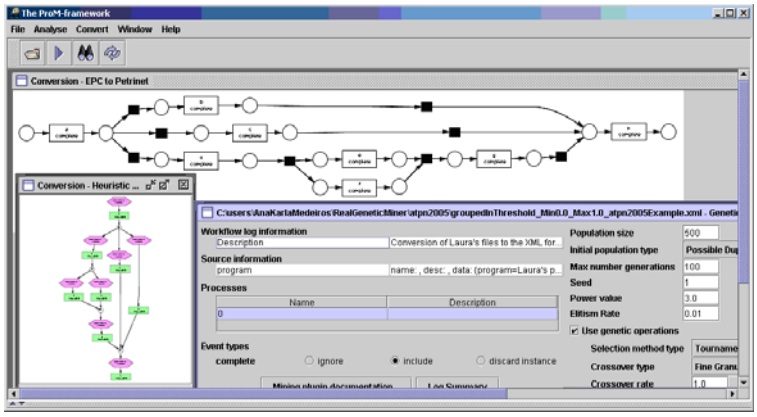


Fig. 14. A screenshot of the GeneticMiner plugin in the ProM framework analyzing the event log in Table 1 and generating the correct process models, i.e., the one shown in Figure 1.

still needed. For instance, the fitness should consider the number of tokens that remained in the individual after the parsing is finished as well as the number of tokens that needed to be added during the parsing. Besides, the dependency relations that are used during the building of the initial population should be modified to become less sensitive to the *missing body* and *missing activity* noise types.

The genetic mining algorithm presented in this paper is supported by a plugin in the ProM framework (cf. http://www.processmining.org). Figure 14 shows a screenshot of the plugin showing the result for the process model with 8 activities in terms of Petri nets and in terms of Event-Driven Process Chains (EPCs). Note that the internal representation used by the GeneticMiner plugin is the causal matrix. However, the ProM framework allows the user to convert this result to other notations such as Petri nets and EPCs.

5 Related Work

The idea of process mining is not new [6, 8, 10–12, 20–22, 26, 28, 39, 40, 5, 42]. Cook and Wolf have investigated similar issues in the context of software engineering processes. In [10] 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 [10] are limited to sequential behavior. Cook and Wolf extend their work to concurrent processes in [11]. 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 Petri net rather than a set of dependency relations between events. In [12] 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 [8]. 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 OR) of joins and splits. As shown in [25], workflow graphs use true and false tokens which do not allow for cyclic graphs. Nevertheless, [8] 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 [28], a tool based on these algorithms is presented. Schimm [39, 40] 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 [21, 20, 22] using an inductive approach. The work presented in [22] is limited to sequential models. The approach described in [21, 20] also allows for concurrency. It uses stochastic activity graphs as an intermediate representation and it generates a workflow model described in the ADONIS modeling 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 [8, 28]. The nature of splits and joins (i.e., AND or OR) 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, our work [26, 42] is characterized by the focus on workflow processes with concurrent behavior (rather than adding ad-hoc mechanisms to capture parallelism). In [42] 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 [42] only provide heuristics and focus on issues such as noise. In [3] 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 [7]. For a detailed explanation of the constructs the α-algorithm does not correctly mine and an extension to mine short-loops, see [29, 30].

Process mining can be seen as a tool in the context of Business (Process) Intelligence (BPI). In [17] a BPI toolset on top of HP's Process Manager is de-

scribed. 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 [14, 34, 35] the design of such warehouse and related issues are discussed in the context of workflow logs. Moreover, [35] 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) [23]. The later tool is commercially available and a customized version of PPM is the Staffware Process Monitor (SPM) [41] 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 [8, 10–12, 20–22, 26, 28, 39, 40, 42].

More from a theoretical point of view, the rediscovery problem discussed in this paper is related to the work discussed in [9, 16, 36]. In these papers the limits of inductive inference are explored. For example, in [16] 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 [9, 16, 36] 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 [5].

There have been some papers combining Petri nets and genetic algorithms, cf. [27, 33, 32, 37]. 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.

6 Conclusion and Future Work

In this paper we presented a new genetic algorithm (i.e. a more global technique) to mine process models. After the introduction of process mining and its practical relevance, we motivated our genetic approach. The use of a genetic

approach seems specially attractive if the event log contains noise. After the introduction of a new process representation formalism (i.e. the causal matrix) and its semantics, we presented the details of our GA: the genetic operators and two fitness measures $Fitness_S$ and $Fitness_C$. Both fitness measures are related to the successful parsing of the material in the event log, but $Fitness_S$ parsing semantics stops when a error occurs. $Fitness_C$ is a more global fitness measure in the sense that its parsing semantics will not stop when an error occurs: the error is registered and the parsing continues.

In the experimental part we presented the results of the genetic process mining algorithm on event logs with and without noise. We specially focused on the performance differences between the two fitness measures (i.e. $Fitness_S$ and $Fitness_C$). The main result is that for both noise-free and noisy event logs, the performance of the GA with the most global fitness measure ($Fitness_C$) appears to be better.

If we look at the performance behavior of the GA for the different noise types (i.e. *missing head*, *missing body*, *missing tail*, *missing activity*, *exchanged activities* and *mixed noise*), we observe special mining problems for the *missing body* and *missing activity* noise types; it happens that they introduce superfluous connections in the process model. A possible solution is an improvement of the fitness measure so that simple process models are preferred above more complex models.

The genetic mining algorithm presented in this paper is supported by a plug-in in the ProM framework (cf. http://www.processmining.org). The reader is encouraged to download the tool and experiment with it (there are also several adaptors for commercial systems). We also invite other research groups to contribute to this initiative by adding additional plugins.

Acknowledgements
The authors would like to thank Boudewijn van Dongen, Peter van den Brand, Minseok Song, Laura Maruster, Eric Verbeek, Monique Jansen-Vullers, and Hajo Reijers for their on-going work on process mining techniques and tools at Eindhoven University of Technology.

References

1. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
2. W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 1–65. Springer-Verlag, Berlin, 2004.
3. 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.

4. 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.
5. 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.
6. W.M.P. van der Aalst and A.J.M.M. Weijters, editors. *Process Mining*, Special Issue of Computers in Industry, Volume 53, Number 3. Elsevier Science Publishers, Amsterdam, 2004.
7. 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.
8. R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In *Sixth International Conference on Extending Database Technology*, pages 469–483, 1998.
9. D. Angluin and C.H. Smith. Inductive Inference: Theory and Methods. *Computing Surveys*, 15(3):237–269, 1983.
10. 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.
11. 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, 1998.
12. 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.
13. J. Dehnert and W.M.P. van der Aalst. Bridging the Gap Between Business Models and Workflow Specifications. *International Journal of Cooperative Information Systems*, 13(3):289–332, 2004.
14. 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.
15. A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Natural Computing. Springer-Verlag, Berlin, 2003.
16. E.M. Gold. Complexity of Automaton Identification from Given Data. *Information and Control*, 37(3):302–320, 1978.
17. 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.
18. K. van Hee, N. Sidorova, and M. Voorhoeve. Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In W.M.P. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 335–354. Springer-Verlag, Berlin, 2003.

19. J. Herbst. A Machine Learning Approach to Workflow Management. In *Proceedings 11th European Conference on Machine Learning*, volume 1810 of *Lecture Notes in Computer Science*, pages 183–194. Springer-Verlag, Berlin, 2000.
20. 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.
21. J. Herbst. *Ein induktiver Ansatz zur Akquisition und Adaption von Workflow-Modellen*. PhD thesis, Universität Ulm, November 2001.
22. 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.
23. IDS Scheer. ARIS Process Performance Manager (ARIS PPM): Measure, Analyze and Optimize Your Business Process Performance (whitepaper). IDS Scheer, Saarbruecken, Gemany, http://www.ids-scheer.com, 2002.
24. G. Keller and T. Teufel. *SAP R/3 Process Oriented Implementation*. Addison-Wesley, Reading MA, 1998.
25. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2003. Available via http://www.workflowpatterns.com.
26. 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.
27. 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.
28. 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.
29. A.K.A. 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.
30. A.K.A. 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.
31. M. Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, 1996.
32. J.H. Moore and L.W. Hahn. Petri net modeling of high-order genetic systems using grammatical evolution. *BioSystems*, 72(1-2):177–86, 2003.
33. 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, EvoWorkshops 2004*, volume 3005 of *Lecture Notes in Computer Science*, pages 63–72. Springer-Verlag, Berlin, 2004.

34. 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.
35. 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.
36. 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.
37. 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.
38. 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.
39. G. Schimm. Process Mining. http://www.processmining.de/.
40. 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.
41. Staffware. Staffware Process Monitor (SPM). http://www.staffware.com, 2002.
42. 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.

A Relating the Causal Matrix and Petri nets

In this section we relate the causal matrix to Petri nets. We will map Petri nets (in particular WF-nets) onto the notation used by our genetic algorithm, i.e., the causal matrix. Then we consider the mapping of the causal matrix onto Petri nets. However, first we introduce some basic notions (e.g., Petri Net, WF-nets, and soundness).

A.1 Preliminaries

This subsection introduces the basic *Petri net* terminology and notations, and also discusses concepts such as *WF-nets* and *soundness*.

The classical Petri net is a directed bipartite graph with two node types called *places* and *transitions*. The nodes are connected via directed *arcs*.

Definition 1 (Petri net). *A Petri net is a triple* (P, T, F)*:*

- P *is a finite set of places,*
- T *is a finite set of transitions* ($P \cap T = \emptyset$)*,*
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation)*

A place p is called an *input place* of a transition t iff there exists a directed arc from p to t. Place p is called an *output place* of transition t iff there exists a directed arc from t to p. *For any relation/directed graph* $G \subseteq N \times N$ we define the preset $\bullet n = \{(m_1, m_2) \in G \mid n = m_2\}$ and postset $n\bullet = \{(m_1, m_2) \in G \mid n = m_1\}$ for any node $n \in N$. We use $\overset{G}{\bullet}\, n$ or $n \overset{G}{\bullet}$ to explicitly indicate the context G if needed. Based on the flow relation F we use this notation as follows. $\bullet t$ denotes the set of input places for a transition t. The notations $t\bullet$, $\bullet p$ and $p\bullet$ have similar meanings, e.g., $p\bullet$ is the set of transitions sharing p as an input place. Note that we do not consider multiple arcs from one node to another.

At any time a place contains zero or more *tokens*, drawn as black dots. The *state*, often referred to as marking, is the distribution of tokens over places, i.e., $M \in P \to \mathbb{N}$. To compare states we define a partial ordering. For any two states M_1 and M_2, $M_1 \leq M_2$ iff for all $p \in P$: $M_1(p) \leq M_2(p)$

The number of tokens may change during the execution of the net. Transitions are the active components in a Petri net: they change the state of the net according to the following *firing rule*:

(1) A transition t is said to be *enabled* iff each input place p of t contains at least one token.

(2) An enabled transition may *fire*. If transition t fires, then t *consumes* one token from each input place p of t and *produces* one token for each output place p of t.

Given a Petri net (P, T, F) and a state M_1, we have the standard notations for a transition t that is enabled in state M_1 and firing t in M_1 results in state M_2 (notation: $M_1 \xrightarrow{t} M_2$) and a firing sequence $\sigma = t_1 t_2 t_3 \ldots t_{n-1}$ leads from state M_1 to state M_n via a (possibly empty) set of intermediate states (notation: $M_1 \xrightarrow{\sigma} M_n$). A state M_n is called *reachable* from M_1 (notation $M_1 \xrightarrow{*} M_n$) iff there is a firing sequence σ such that $M_1 \xrightarrow{\sigma} M_n$. Note that the empty firing sequence is also allowed, i.e., $M_1 \xrightarrow{*} M_1$.

In this appendix, we will focus on a particular type of Petri nets called *WorkFlow nets* (WF-nets) [1, 2, 13, 18].

Definition 2 (WF-net). *A Petri net $PN = (P, T, F)$ is a WF-net (Workflow net) if and only if:*

(i) There is one source place $i \in P$ such that $\bullet i = \emptyset$.

(ii) There is one sink place $o \in P$ such that $o\bullet = \emptyset$.

(iii) Every node $x \in P \cup T$ is on a path from i to o.

A WF-net represents the life-cycle of a case that has some initial state represented by a token in the unique input place (i) and a desired final state represented by a token in the unique output place (o). The third requirement in Definition 2 has been added to avoid "dangling transitions and/or places". In the context of workflow models or business process models, transitions can be interpreted as *activities* or *tasks* and places can be interpreted as *conditions*. Although the term "WorkFlow net" suggests that the application is limited to

workflow processes, the model has wide applicability, i.e., any process where each case has a life-cycle going from some initial state to some final state fits this basic model.

The three requirements stated in Definition 2 can be verified statically, i.e., they only relate to the structure of the Petri net. To characterize desirable dynamic properties, the notation of *soundness* has been defined [1, 2, 13, 18].

Definition 3 (Sound). *A procedure modeled by a WF-net $PN = (P, T, F)$ is sound if and only if:*

(i) For every state M reachable from state i, there exists a firing sequence leading from state M to state o. Formally: $\forall_M (i \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} o)$.[3]

(ii) State o is the only state reachable from state i with at least one token in place o. Formally: $\forall_M (i \xrightarrow{*} M \ \wedge \ M \geq o) \Rightarrow (M = o)$.

(iii) There are no dead transitions in (PN, i). Formally: $\forall_{t \in T} \ \exists_{M,M'} \ i \xrightarrow{*} M \xrightarrow{t} M'$.

Note that the soundness property relates to the dynamics of a WF-net. The first requirement in Definition 3 states that starting from the initial state (state i), it is always possible to reach the state with one token in place o (state o). The second requirement states that the moment a token is put in place o, all the other places should be empty. The last requirement states that there are no dead transitions (activities) in the initial state i.

A.2 Mapping a Petri net onto a Causal Matrix

In this paper, we use the concept of a *causal matrix* to represent an individual. Table 3 and Table 2 show two alternative visualizations. In this section, we first formalize the notion of a causal matrix. This formalization will be used to map a causal matrix onto a Petri net and vice versa.

Definition 4 (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,*
- *$I \in A \rightarrow \mathcal{P}(\mathcal{P}(A))$ is the input condition function,*[4]
- *$O \in A \rightarrow \mathcal{P}(\mathcal{P}(A))$ is the output condition function,*

such that

- $C = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup I(a_2)\}$,[5]
- $C = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup O(a_1)\}$,

[3] Note that there is an overloading of notation: the symbol i is used to denote both the *place* i and the *state* with only one token in place i.

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

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

- $\forall_{a \in A}\ \forall_{s,s' \in I(a)}\ s \cap s' \neq \emptyset\ \Rightarrow\ s = s'$,
- $\forall_{a \in T}\ \forall_{s,s' \in O(a)}\ s \cap s' \neq \emptyset\ \Rightarrow\ s = s'$,
- $C \cup \{(a_o, a_i) \in A \times A \mid a_o \overset{C}{\bullet} = \emptyset \wedge \overset{C}{\bullet} a_i = \emptyset\}$ *is a strongly connected graph.*

The mapping of Table 3 onto $CM = (A, C, I, O)$ is straightforward (the latter two columns represent I and O). Note that C can be derived from both I and O. Its main purpose is to ensure consistency between I and O. For example, if a_1 has an output condition mentioning a_2, then a_2 has an input condition mentioning a_1 (and vice versa). This is enforced by the first two constraints. The third and fourth constraint indicate that some activity a may appear only once in the conjunction of disjunctions, e.g., $\{\{A, B\}, \{A, C\}\}$ is not allowed because A appears twice. The last requirement has been added to avoid that the causal matrix can be partitioned in two independent parts or that nodes are not on a path from some source activity a_i to a sink activity a_o.

The mapping from an arbitrary Petri net to its corresponding causal matrix illustrates the expressiveness of the internal format used for genetic mining. First, we give the definition of the mapping $\Pi_{PN \to CM}$.

Definition 5 ($\Pi_{PN \to CM}$). *Let $PN = (P, T, F)$ be a Petri net. $\Pi_{PN \to CM}(PN) = (A, C, I, O)$, i.e., the mapping of PN, where*

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

Let PN be the Petri net shown in Figure 1. It is easy to check that $\Pi_{PN \to CM}(PN)$ is indeed the causal matrix in Table 2. However, there may be Petri nets PN for which $\Pi_{PN \to CM}(PN)$ is not a causal matrix. The following lemma shows that for the class of nets we are interested in, i.e., WF-nets, the requirement that there may not be two different places in-between two activities is sufficient to prove that $\Pi_{PN \to CM}(PN)$ represents a causal matrix as defined in Definition 4.

Lemma 1. *Let $PN = (P, T, F)$ be a WF-net with no duplicate places in between two transitions, i.e., $\forall_{t_1, t_2 \in T} |t_1 \bullet \cap \bullet t_2| \leq 1$. $\Pi_{PN \to CM}(PN)$ represents a causal matrix as defined in Definition 4.*

Proof. Let $\Pi_{PN \to CM} = (A, C, I, O)$. Clearly, $A = T$ is a finite set, $C \subseteq A \times A$, and $I, O \in A \to \mathcal{P}(\mathcal{P}(A))$. $C = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup I(a_2)\}$ because $a_1 \in \bigcup I'(a_2)$ if and only if $a_1 \bullet \cap \bullet a_2 \neq \emptyset$. Similarly, $C = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup O(a_1)\}$. $\forall_{a \in A}\ \forall_{s,s' \in I(a)}\ s \cap s' \neq \emptyset\ \Rightarrow\ s = s'$ because $\forall_{t_1, t_2 \in T}\ |t_1 \bullet \cap \bullet t_2| \leq 1$. Similarly, $\forall_{a \in A}\ \forall_{s,s' \in O(t)}\ s \cap s' \neq \emptyset\ \Rightarrow\ s = s'$. Finally, it is easy to verify that $C \cup \{(a_o, a_i) \in A \times A \mid a_o \bullet = \emptyset\ \wedge\ \bullet a_i = \emptyset\}$ is a strongly connected graph. □

The requirement $\forall_{t_1, t_2 \in T}\ |t_1 \bullet \cap \bullet t_2| \leq 1$ is a direct result of the fact that in the conjunction of disjunctions in I and O, there may not be any overlaps. This restriction has been added to reduce the search space of the genetic mining algorithm, i.e., the reason is more of a pragmatic nature. However, for the success of the genetic mining algorithm such reductions are of the utmost importance.

A.3 A Naive Way of Mapping a Causal Matrix onto a Petri net

The mapping from a causal matrix onto a Petri net is more involved because we need to "discover places" and, as we will see, the causal matrix is slightly more expressive than classical Petri nets.[6] Let us first look at a naive mapping.

Definition 6 ($\Pi^N_{CM \to PN}$). *Let $CM = (A, C, I, O)$ be a causal matrix. $\Pi^N_{CM \to PN}(CM) = (P, T, F)$, i.e., the naive Petri net mapping of CM, where*

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

The mapping $\Pi^N_{CM \to PN}$ maps activities onto transitions and adds input places and output places to these transitions based on functions I and O. These places are local to one activity. To connect these local places, one transition m_{t_1,t_2} is added for every $(t_1, t_2) \in C$. Figure 15 shows a causal matrix and the naive mapping $\Pi^N_{CM \to PN}$ (we have partially omitted place/transition names).

ACTIVITY	*INPUT*	*OUTPUT*
A	$\{\}$	$\{\{C, D\}\}$
B	$\{\}$	$\{\{D\}\}$
C	$\{\{A\}\}$	$\{\}$
D	$\{\{A, B\}\}$	$\{\}$

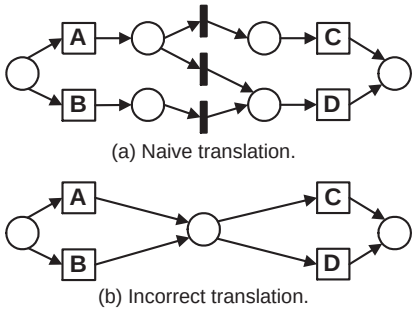


Fig. 15. A causal matrix (left) and two potential mappings onto Petri nets (right).

Figure 15 shows two WF-nets illustrating the need for "silent transitions" of the form m_{t_1,t_2}. The dynamics of the WF-net shown in Figure 15(a) is consistent with the causal matrix. If we try to remove the silent transitions, the best candidate seems to be the WF-net shown in Figure 15(b). Although this is a sound WF-net capturing incorporating the behavior of the WF-net shown in Figure 15(a), the mapping is *not* consistent with the causal matrix. Note that Figure 15(b) allows for a firing sequence where B is followed by C. This does not make sense because $C \notin \bigcup O(B)$ and $B \notin \bigcup I(C)$. Therefore, we use the mapping given in Definition 6 to give Petri-net semantics to causal matrices.

It is easy to see that a causal matrix defines a WF-net. However, note that the WF-net does not need to be sound.

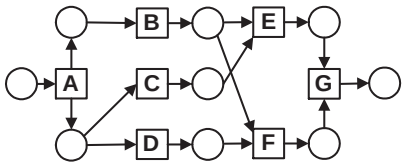
[6] Expressiveness should not be interpreted in a formal sense but in the sense of convenience when manipulating process instances, e.g., crossover operations.

Lemma 2. *Let $CM = (A, C, I, O)$ be a causal matrix. $\Pi^N_{CM \to PN}(CM)$ is a WF-net.*

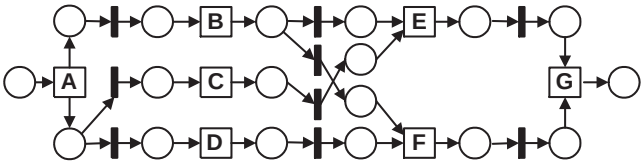
Proof. It is easy to verify the three properties mentioned in Definition 2. Note that the "short-circuited" C is strongly connected and that each m_{t_1,t_2} transition makes a similar connection in the resulting Petri net. □

Figure 16 shows that despite the fact that $\Pi^N_{CM \to PN}(CM)$ is a WF-net, the introduction of silent transitions may introduce a problem. Figure 16(b) shows the WF-net based on Definition 6, i.e., the naive mapping. Clearly, Figure 16(b) is not sound because there are two potential deadlocks, i.e., one of the input places of E is marked and one of the input places of F is marked but none of them is enabled. The reason for this is that the choices introduced by the silent transitions are not "coordinated" properly. If we simply remove the silent transitions, we obtain the WF-net shown in Figure 16(a). This network is consistent with the causal matrix. This can easily be checked because applying the mapping $\Pi_{PN \to CM}$ defined in Definition 5 to this WF-net yields the original causal matrix shown in Figure 16.

ACTIVITY	*INPUT*	*OUTPUT*
A	$\{\}$	$\{\{B\},\{C,D\}\}$
B	$\{\{A\}\}$	$\{\{E,F\}\}$
C	$\{\{A\}\}$	$\{\{E\}\}$
D	$\{\{A\}\}$	$\{\{F\}$
E	$\{\{B\},\{C\}\}$	$\{\{G\}\}$
F	$\{\{B\},\{D\}\}$	$\{\{G\}\}$
G	$\{\{E\},\{F\}\}$	$\{\}$



(a) Mapping without silent transitions.



(b) Naive mapping.

Fig. 16. Another causal matrix (left) and two potential mappings onto Petri nets (right).

Figures 15 and 16 show a dilemma. Figure 15 demonstrates that silent transitions are needed while Figure 16 proves that silent transitions can be harmful. There are two ways to address this problem taking the mapping of Definition 6 as a starting point.

First of all, we can use *relaxed soundness* [13] rather than soundness [1]. This implies that we only consider so-called sound firing sequences and thus avoid the two potential deadlocks in Figure 16(b). See [13] for transforming a relaxed sound WF-net into a sound one.

Second, we can change the firing rule such that silent transitions can only fire if they actually enable a non-silent transition. The enabling rule for non-silent transitions is changed as follows: *a non-silent transition is enabled if each of its input places is marked or it is possible to mark all input places by just firing silent transitions*, i.e., silent transitions only fire when it is possible to enable a non-silent transition. Note that non-silent and silent transitions alternate and therefore it is easy to implement this semantics in a straightforward and localized manner.

In this appendix we use the second approach, i.e., a slightly changed enabling/-firing rule is used to specify the semantics of a causal matrix in terms of a WF-net. This semantics allows us also to define a notion of fitness required for the genetic algorithms. Using the Petri-net representation we can play the "token game" to see how each event trace in the log fits the individual represented by a causal matrix. Note that in Figure 3, the token game is played in the context of the causal matrix. However, the semantics used is exactly the same.

A.4 A More Sophisticated Mapping

Although not essential for the genetic algorithms, we elaborate a bit on the dilemma illustrated by figures 15 and 16. The dilemma shows that the causal net representation is slightly more expressive than ordinary Petri nets. (Note the earlier comment on expressiveness!) Therefore, it is interesting to see which causal matrices can be directly mapped onto a WF-net without additional silent transitions. For this purpose we first define a mapping $\Pi^R_{CM \to PN}$ which only works for a restricted class of causal matrices.

Definition 7 ($\Pi^R_{CM \to PN}$). *Let $CM = (A, C, I, O)$ be a causal matrix. $\Pi^R_{CM \to PN}(CM) = (P, T, F)$, i.e., the restricted Petri net mapping of CM, where*

- $X = \{(T_i, T_o) \in \mathcal{P}(A) \times \mathcal{P}(A) \mid \forall_{t \in T_i}\ T_o \in O(t)\ \wedge\ \forall_{t \in T_o}\ T_i \in I(t)\}$
- $P = X \cup \{i, o\}$,
- $T = A$,
- $F = \{(i, t) \mid t \in T \wedge \overset{C}{\bullet} t = \emptyset\} \cup \{(t, o) \mid t \in T\ \wedge\ t \overset{C}{\bullet} = \emptyset\} \cup \{((T_i, T_o), t) \in X \times T \mid t \in T_o\} \cup \{(t, (T_i, T_o)) \in T \times X \mid t \in T_i\}$.

If we apply this mapping to the causal matrix shown in Figure 16, we obtain the WF-net shown in Figure 16(a), i.e., the desirable net without the superfluous silent transitions. However, in some cases the $\Pi^R_{CM \to PN}$ does not yield a WF-net because some connections are missing. For example, if we apply $\Pi^R_{CM \to PN}$ to the causal matrix shown in Figure 15, then we obtain a result where there are no connections between A, B, C, and D. This makes sense because there does not exist a corresponding WF-net. This triggers the question whether it is possible to characterize the class of causal matrices for which $\Pi^R_{CM \to PN}$ yields the correct WF-net.

Definition 8 (Simple). *Let $CM = (A, C, I, O)$ be a causal matrix. CM is simple if and only if* $\forall_{t_A, t_B \in T}\ \forall_{T_A \in O(t_A)}\ \forall_{T_B \in O(t_B)}\ \forall_{t_C \in (T_A \cap T_B)}\ \forall_{T_C \in I(t_C)} \{t_A, t_B\} \subseteq$

$T_C \Rightarrow T_A = T_B$ *and* $\forall_{t_A,t_B \in T}\ \forall_{T_A \in I(t_A)}\ \forall_{T_B \in I(t_B)}\ \forall_{t_C \in (T_A \cap T_B)}\ \forall_{T_C \in O(t_C)}$ $\{t_A, t_B\} \subseteq T_C \Rightarrow T_A = T_B$.

Figure 17 illustrates the notion of simplicity. Note that the diagrams shown do not represent Petri nets but causal matrices, i.e., the circles should not be interpreted as places but as disjunctions. (Recall that the input and output sets of an activity are conjunctions of disjunctions.) The left-hand side shows a requirement on two activities (t_A and t_B) pointing to the same activity (t_C). In this case, T_A and T_B should coincide. The left-hand side of Figure 17 shows the requirement on two activities (t_A and t_B) pointed to by the same activity (t_C). In this case, T_A and T_B should also coincide.

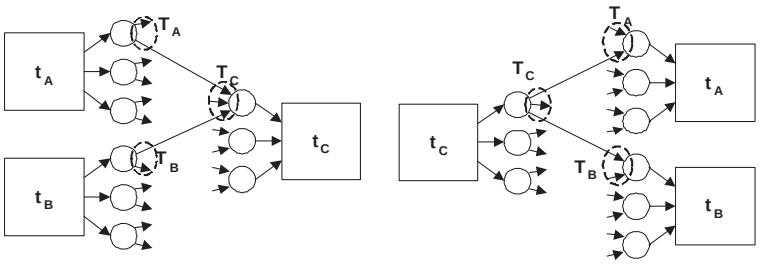


Fig. 17. A causal matrix is simple if in the situations shown $T_A = T_B$. If one can find a pattern like one of the two shown and $T_A \neq T_B$, then the causal matrix is not simple.

Clearly the causal matrix shown in Figure 16 is simple while the one in Figure 15 is not. In the remainder of this appendix we will show that $\Pi^R_{CM \to PN}$ provides indeed the correct mapping if the causal matrix is simple.

Lemma 3. *Let $CM = (A, C, I, O)$ be a causal matrix. If CM is simple, then each of the following properties holds:*

(i) $\forall_{(t_1,t_2) \in C}\ \exists_{T_1,T_2 \in \mathcal{P}(A)}\ t_1 \in T_1 \wedge t_2 \in T_2 \wedge (\forall_{t \in T_1}\ T_2 \in O(t)) \wedge (\forall_{t \in T_2}\ T_1 \in I(t))$*, and*

(ii) $\Pi^R_{CM \to PN}(CM)$ *is a WF-net.*

Proof. First, we prove Property (i). Assume two activities t_1 and t_2 such that $(t_1, t_2) \in C$. There is exactly one $T_2 \in O(t_1)$ such that $t_2 \in T_2$ because $(t_1, t_2) \in C$ and disjunctions cannot overlap. Similarly, there is exactly one $T_1 \in I(t_2)$ such that $t_1 \in T_1$. Remains to prove that $\forall_{t \in T_1}\ T_2 \in O(t)$ and $\forall_{t \in T_2}\ T_1 \in I(t)$. Suppose $t \in T_1$. Clearly, there is an $X \in O(t)$ such that $t_2 \in X$. Now we can apply the left-hand side of Figure 17 with $t_A = t_1$, $t_C = t_2$, $t_B = t$, $T_A = T_2$, $T_C = T_1$, and $T_B = X$. This implies that $T_2 = X$ and hence $\forall_{t \in T_1}\ T_2 \in O(t)$. Similarly, we can show that $\forall_{t \in T_2}\ T_1 \in I(t)$.

The second property (Property (ii)) follows from the first one because if $(t_1, t_2) \in C$ then a connecting place between t_1 and t_2 is introduced by the set

X used in the construction of $\Pi^R_{CM \to PN}(CM)$ (cf. Definition 7). The rest of the proof is similar to the proof of Lemma 2. □

The two properties given in Lemma 3 are used to prove that mapping a causal matrix to a Petri net using $\Pi^R_{CM \to PN}(CM)$ and then applying the mapping of a Petri net to a causal matrix $\Pi_{PN \to CM}$ on the result, yields again the original causal matrix.

Theorem 1. *Let $CM = (A, C, I, O)$ be a causal matrix. If CM is simple, then $\Pi_{PN \to CM}(\Pi^R_{CM \to PN}(CM)) = CM$.*

Proof. Consider an activity t in CM with input sets $I(t)$ and output sets $O(t)$. Every $p \in I(t)$ is mapped onto an input place in $\Pi^R_{CM \to PN}(CM)$ bearing the label (T_i, T_o) such that $p = T_i$. Lemma 3 can be used to show that such a place exists if CM is simple. Every $p \in O(t)$ is mapped onto an output place in $\Pi^R_{CM \to PN}(CM)$ bearing the label (T_i, T_o) such that $p = T_o$. Again, Lemma 3 can be used to show that such a place exists. Therefore, no information is lost during the mapping onto the WF-net $\Pi^R_{CM \to PN}(CM)$ and that $\Pi_{PN \to CM}$ retranslates the sets T_i and T_o in the places of X to functions I and O. Hence the original causal matrix is reconstructed. □

In this appendix, we discussed the relation between the representation used by our genetic algorithm and Petri nets. We used this relation to give semantics to our representation. It was shown that this representation is slightly more expressive than Petri nets because any WF-net can be mapped into causal matrix while the reverse is only possible after introducing silent transitions and modifying the firing rule or using relaxed soundness. We also characterized the class of causal matrices that can be mapped directly.