Genetic Process Mining: A Basic Approach and its Challenges

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. One of the aims of process mining is to retrieve a process model from a given event log. However, current techniques have problems when mining processes that contain non-trivial constructs and/or when dealing with the presence of noise in the logs. To overcome these problems, we try to use genetic algorithms to mine process models. The non-trivial constructs are tackled by choosing an internal representation that supports them. The noise problem is naturally tackled by the genetic algorithm because, per definition, these algorithms are robust to noise. The definition of a good fitness measure is the most critical challenge in a genetic approach. This paper presents the current status of our research and the pros and cons of the fitness measure that we used so far. Experiments show that the fitness measure leads to the mining of process models that can reproduce all the behavior in the log, but these mined models may also allow for extra behavior. In short, the current version of the genetic algorithm can already be used to mine process models, but future research is necessary to always ensure that the mined models do not allow for extra behavior. Thus, this paper also discusses some ideas for future research that could ensure that the mined models will *always* only reflect the behavior in the log.

Keywords: process mining, genetic mining, genetic algorithms, Petri nets, workflow Petri nets.

1 Introduction

One of the aims of process mining is to *automatically* build a process model that describes the behavior contained in an event log. The models mined by process mining tools can be used as an objective starting point during the deployment of systems that support the execution of processes and/or as a feedback mechanism to check the prescribed process model against the enacted one. We illustrate how process mining techniques work using an example. Consider the event log shown in Table 1. This log shows the events for applying to get a license to ride motorbikes or drive cars. Note that applicants for different types of licenses do the same theoretical exam (task C) but different practical ones (tasks D or E). In other words, whenever task B is executed, task E is the only one that can be executed after the applicant has done the theoretical exam. This shows that there is a dependency between tasks B and E, and also between tasks A and D. Based on this log and these observations, process mining tools could retrieve the model in Figure 1. In this case, we are using Petri nets to depict this model. We do so because Petri nets [6] will be used to explain how the semantics of our internal representation work.

ID	Process instance	ID	Process instance	ID	Process instance
1	X, A, C, D, Y	3	X, A, C, D, Y	5	X, B, C, E, Y
2	X, B, C, E, Y	4	X, B, C, E, Y	6	X, A, C, D, Y

Table 1. Where: X = "Apply for license", A = "Attend to classes on how to ride motorbikes", B = "Attend to classes on how to drive cars", C = "Do theoretical exam", D = "Do practical exam to ride motorbikes", E = "Do practical exam to drive cars", and Y = "Get result".

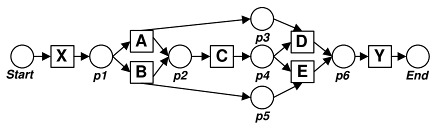


Fig. 1. Mined net for the log in Table 1

Petri nets are a formalism to model concurrent processes. Graphically, Petri nets are bipartite directed graphs with two node types: *places* and *transitions*. Places represent conditions in the process. Transitions represent actions. Tasks in the event logs correspond to transitions in Petri nets. The state of a Petri net (or process for us) is described by adding tokens (black dots) to places. The dynamics of the Petri net is determined by the *firing rule*. A transition can be executed (i.e. an action can take place in the process) when all of its input places (i.e. pre-conditions) have at least a number of tokens that is equal to the number of directed arcs from the place to the transition. After execution, the transition removes tokens from the input places (one token is removed for every input arc from the place to the transition) and produces tokens for the output places (again, one token is produced for every output arc). Besides, the Petri nets that we consider have a single *start* place and a single *end* place. This means that the processes we describe have a single start point and a single end point. For the Petri net in Figure 1, in the initial state there is only one token in place *Start*. This implies that X is the only transition that can be executed in the initial state.

Current research in process mining [1, 4] still has problems to discover process models with certain structural constructs and/or to deal with the presence of noise in the logs. The problematic constructs are: *non-free-choice*, *invisible tasks* and *duplicate tasks* [4]. Non-free-choice constructs combine synchronization and choice. The example in Figure 1 illustrates a non-free-choice construct involving the tasks D and E. The current techniques do not capture the dependency between the tasks $A - D$ and $B - E$. Invisible tasks are only used for routing purposes (for instance, to skip the execution of another task) and do not appear in the log. The current techniques do not mine models with unlabelled tasks. Duplicate tasks mean that multiple transitions have the same label in the original process model. The problem here is that most of the mining techniques treat these duplicate tasks as a single one. *Noise* can appear in two situations: event traces were somehow incorrectly logged or event traces reflect exceptional situations. Either way, most of the techniques will try to find a process model that can parse all the traces in the log. However, the presence of noise will hinder the correct mining of the most common behavior.

One of the reasons why the current techniques cannot completely cope with the above mentioned problematic constructs and/or with noisy logs is because their search is based on *local* information in the log. For instance, the α-algorithm (see [2] for details) uses only information about which tasks directly succeed or precede one another in the process instances. As a result, this algorithm does not capture the dependency in non-free-choice constructs. For example, the α-algorithm will never discover the Petri net in Figure 1 for the log in Table 1 because none of the process instances has the sub-trace "A,D" or "B,E". Consequently, the α-algorithm will not link these tasks. To overcome these problems, our research uses *genetic algorithms* [3] to mine process models. The main motivation is to benefit from the *global* search that is performed by this kind of algorithms.

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

When using genetic algorithms to mine process models, there are three main concerns. The first is to define the *internal representation.* The internal representation defines the search space of a genetic algorithm. The internal representation that we define and explain in this paper supports all the problematic constructs, except for duplicate tasks. The second concern is to define the *fitness measure.* In our case, the fitness measure evaluates the quality of a point (individual or process model) in the search space against the event log. A genetic algorithm searches for individuals whose fitness is 100%. Thus, our fitness measure makes sure that individuals with a 100%-fitness will parse all the process instances (traces) in the log. The third concern relates to the *genetic operators* (crossover and mutation) because they should ensure that all points in the search space defined by the internal representation may be reached when the genetic algorithm runs. This paper presents a genetic algorithm that addresses these three concerns.

The rest of the paper is organized as follows. Section 2 explains the internal representation that we use and its semantics. Section 3 explains how the genetic algorithm works. Section 4 discusses the experiments and results. This section also shows the results when the genetic algorithm uses some heuristics during the building of the initial population. Section 5 presents the current ideas to make sure that the returned model is "minimal". Section 6 presents the conclusions and future work.

2 Internal Representation and Semantics

When defining the internal representation to be used by our genetic algorithm, the main requirement was that this representation should express the dependencies between the tasks in the log. In other words, the model should clearly express which tasks would enable the execution of other tasks. Additionally, it would be nice if the internal representation would be compatible with a formalism to which analysis techniques and tools exist. This way, these techniques could also be applied to the discovered models. Thus, one option would be to directly represent the individual (or process model) as a Petri net [6]. However, such a representation would require determining the number of places in every individual and this is not the core concern. It is more important to show the dependencies between the tasks. So, we defined an internal representation that is as expressive as Petri nets (from the task dependency perspective) but that focusses only on the dependencies between tasks. This representation is called *causal matrix*. Figure 2 illustrates in (i) the causal matrix[1] that expresses the same task dependencies that are in the "original Petri net". The causal matrix shows which tasks enable the execution of other tasks via the matching of *input* (I) and *output* (O) condition functions. The sets returned by the condition functions I and O have *subsets* that contain the tasks in the model. Tasks in a same subset have an OR-split/join relation. Sets in different subsets have an AND-split/join relation. Thus, every I and O set expresses a conjunction of disjunctions. Additionally, a task may appear in more than one subset in a same set. As an example, for task D in the original Petri net in Figure 2 the causal matrix states that $I(D) = \{\{F, B, E\}, \{E, C\}, \{G\}\}$ because D is enabled by an AND-join construct that has 3 places. From top to bottom, the first place has a token whenever F *or* B *or* E fires. The second place, whenever E *or* C fires. The third place, whenever G fires. Similarly, the causal matrix has $O(D) = \{\}$ because D is executed last in the model. The following definition formally defines these notions.

Definition 1 (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* [2]*, and $O \in A \rightarrow \mathcal{P}(\mathcal{P}(A))$ is the output condition function; such that: $C = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup I(a_2)\}$* [3]*, $C = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup O(a_1)\}$, and $C \cup \{(a_o, a_i) \in A \times A \mid a_o \overset{C}{\bullet} = \emptyset \wedge \overset{C}{\bullet} a_i = \emptyset\}$ is a strongly connected graph.*

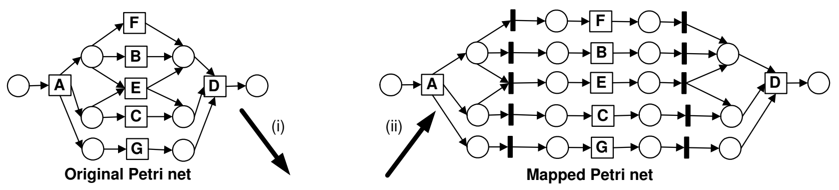
Any Petri net without duplicate tasks and without more than one place with the same input tasks and the same output tasks can be mapped to a causal matrix. The main idea is that there is a causal relation C between any two tasks t and t' whenever at least one of the *output* places of t is an *input*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

The semantics of the causal matrix can be easily understood by mapping them back to Petri nets. *Conceptually*, the causal matrix behaves as a Petri net that contains visible and invisible tasks. For instance, see Figure 2. This figure shows (i) the mapping of a Petri net to a causal matrix and (ii) the mapping from the causal matrix to a Petri net. The firing rule for the mapped Petri net is very similar to the firing rule of Petri nets in general. The only difference concerns the invisible tasks. Enabled invisible tasks can only fire if their firing enables a visible

[1] Due to space limitations, Figure 2 shows a compact representation of this causal matrix.

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

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



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

Compact representation of the causal matrix

Fig. 2. Mapping of a PN with more than one place between two tasks (or transitions).

task. Similarly, a visible task is enabled if all of its input places have tokens or if there exits *a set of* invisible tasks that are enabled and whose firing will lead to the enabling of the visible task. Conceptually, the causal matrix keeps track of the distribution of tokens at a marking in the output places of the visible tasks. Every causal matrix starts with a token at the start place. We point out that, in Figure 2, although the mapped Petri net does not have the same *structure* of the original Petri net, these two nets are *behaviorally* equivalent. In other words, given that these two nets initially have a single token and this token is at the start place (i.e., the input place of A), the set of traces the two nets can generate is the same. Additionally, the invisible tasks in the mapped Petri net show that the causal matrix supports the representation of invisible tasks that are used, for instance, to skip tasks. A detailed explanation and formalization about the mappings in this section can be found in [5].

3 Genetic Algorithm

In this section we describe the main building blocks of our genetic algorithm.

3.1 Initial Population

The initial population is randomly built by the genetic algorithm. As explained in Section 2, individuals are causal matrices. When building the initial population, we roughly follow Definition 1. Given a log, all individuals in any population of the genetic algorithm have the same set of activities (or tasks) A. This set contains the tasks that appear in the log. However, the causality relation C and the condition functions I and O are randomly built for every individual in the population. As a result, the initial population can have any individual in the search space defined by a set of activities A. Note that the higher the amount of tasks that a log contains, the bigger this search space.

3.2 Fitness Calculation

Our fitness is based on the parsing of the event traces by individuals. For a noisy-free log, the perfect individual should have fitness 1. For a noisy log, the perfect individual should have fitness close to 1 (since it would be able to parse most of the traces but it would have problems to parse the noisy traces). From this discussion, a natural fitness for an individual to a given log seems to be the number of properly parsed event traces[4] divided by the total number of event traces. However, this fitness measure is too coarse because it does not give indication about how many parts of an individual are correct when the individual does not properly parse an event trace. So, we defined a more elaborate fitness function: when the task to be parsed is not enabled, the problems (e.g. number of missing tokens to enable this task) are registered and the parsing proceeds as if this task would be enabled. This *continuous* parsing semantic is more robust because it gives a better indication of how many tasks do or do not have problems during the parsing of a trace. The fitness function that our algorithm uses is in Definition 2. The notation used in this definition is as follows. $allParsedActivities(L, CM)$ gives the total number of tasks in the event log L that could be parsed without problems by the causal matrix (or individual) CM. $numActivitiesLog(L)$ gives the number of tasks in L. $allMissingTokens(L, CM)$ indicates the number of missing tokens in all event traces. $allExtraTokensLeftBehind(L, CM)$ indicates the number of tokens that were not consumed after the parsing stopped plus the number of tokens of the end place minus 1 (because of proper completion). $numTracesLog(L)$ indicates the number of traces in L. $numTracesMissingTokens(L, CM)$ and $numTracesExtraTokensLeftBehind(L, CM)$ respectively indicate the number of traces in which tokens were missing or tokens were left behind during the parsing.

Definition 2 (Fitness). *Let L be an event log. Let CM be a causal matrix. Then the fitness $F : L \times CM \rightarrow (-\infty, 1]$ is a function defined as*

$$F(L, CM) = \frac{allParsedActivities(L,CM) - punishment}{numActivitiesLog(L)}, \text{ where}$$

$$punishment = \frac{allMissingTokens(L,CM)}{numTracesLog(L) - numTracesMissingTokens(L,CM)+1} + \frac{allExtraTokensLeftBehind(L,CM)}{numTracesLog(L) - numTracesExtraTokensLeftBehind(L,CM)+1}$$

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

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

ones. In other words, this individual is a better candidate to produce offsprings for the next population (see Subsection 3.4).

3.3 Stop Criteria

The mining algorithm stops when (i) it finds an individual with fitness equals 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.

3.4 Genetic Operators

We use *elitism*, *crossover* and *mutation* to build the individuals of the next generation. A percentage of the best individuals (the *elite*) is directly copied to the next population. The other individuals in the population are generated via crossover and mutation. Two parents produce two offsprings. To select parents, a *tournament* is played in which five individuals in the population are randomly drawn and the fittest one always wins. The *crossover rate* determines the probability that two parents undergo crossover. Crossover is a genetic operator that aims at recombining existing material in the current population. In our case, this material is the set of current causality relations in the population. The crossover operation should allow for the complete search of the space defined by the existing causality relation in a population. Given a set of causality relations, the search space contains all the individuals that can be created by any combination of a subset of the causality relations in the population. Thus, our crossover operator allows an individual to: lose tasks from the subsets in its I/O condition functions (but not necessarily causality relations because a same task may be in more than one subset of an I/O condition function), add tasks to the subsets in its I/O condition functions (again, not necessarily causality relations), exchange causality relations with other individuals, incorporate causality relations that are in the population but are not in the individual, lose causality relations, decrease the number of subsets in its I/O condition functions, *and/or* increase the number of subsets in its I/O condition functions. The crossover point of two parents is a randomly chosen task. Note that, after crossover, the number of causality relations for the whole population remains constant, but how these relations appear in the offsprings may be different from the parents.

After the crossover, the mutation operator takes place. The mutation operator aims at inserting new material in the current population. In our case, this means that the mutation operator may change the existing causality relations of a population. Thus, our mutation operator performs one of the following actions to the I/O condition functions of a task in an individual: (i) randomly choose a subset and add a task (in A) to this subset, (ii) randomly choose a subset and remove a task out of this subset, *or* (iii) randomly redistribute the elements in the subsets of I/O into new subsets. Every task in an offspring may undergo mutation with the probability determined by the *mutation rate*.

4 Experiments and Results

As a first test for our genetic algorithm (GA), we applied it for *noise-free* event logs and checked if it could mine process models that contain *all* the behavior in these logs. In other words, the mined model should have the fitness $F = 1$. During the experiments, the genetic algorithm mined event logs from nets that contain 5, 7, 8, 12 and 22 tasks. These nets contain short loops, parallelism and/or non-free-choice constructs. Every event log has 1000 random executions of the nets. For each noise-free event-log, 10 runs of the genetic algorithm were executed. The populations had 500 individuals and were iterated for at most 100000 generations. The crossover rate was 1.0 and the mutation rate was 0.01. The elitism rate was 0.01. The initial population might contain duplicate individuals. All the experiments were run using the ProM framework, our tool set that can be obtained via *www.processmining.org.* We implemented the genetic algorithm described in this paper as a plug-in for this framework. This framework also supports a mapping from the internal representation to Petri nets (cf. Section 2).

The results in the grey columns of Table 2 show that the GA could find an individual that can parse all log traces in most of the runs [5]. However, none of these individuals are equal to the original nets that were used to generate the event logs. This happens because, although the requirements that the fitness F captures are all *necessary* to ensure that the GA mines a process model that can parse all traces in the log, these requirement are not *sufficient* to ensure that the mined model will *always* give a good insight about what is happening in the log. The reason is that different models are able to parse all event traces and these models may allow for extra behavior that does not belong to the class of traces in the log.

# Tasks	Figure	# Runs $F = 1$		BFE		WFE		MBF		Mean # Generations		Original Found	
5	–	9	10	1	1	0.989	1	0.998	1	5016	2	0	2
7	2	6	10	1	1	0.987	0.999	0.997	0.999	29259	16323	0	4
7 (nfc)	1	10	10	1	1	1	1	1	1	511	0	0	0
8	–	10	10	1	1	1	1	1	1	5145	2	0	9
12	–	10	10	1	1	1	1	1	1	1831	1	0	10
22	–	0	1	0.931	1	0.537	0.972	0.739	0.989	249	192	0	0

Table 2. Results of the mining for 10 runs. "#" means "number of". The columns BFE, WFE and MBF respectively show the Best Fitness Ever, the Worst Fitness Ever (i.e. the best one in the worst run) and the Mean Best Fitness (i.e. average over 10 runs).

Thus, the challenge we have now for our GA is: "How to ensure that the retrieved model that can parse all the traces does not allow for extra undesired behavior as well?". To illustrate this we consider the nets shown in Figure 3. These models can also parse the traces in Table 1, but they allow for extra behavior. For instance, both models allow for the applicant to take the exam before attending to classes. To define a fitness measure to punish models that express more than it is in the log is especially difficult because we do not have negative examples to

[5] Note: The experiments for the log of the net with 22 tasks were run for at most 250 generations because they take too much time to complete. However, the obtained results suggest that the population was evolving towards the right direction. Due to space limitation, the nets with 5, 8, 12 and 22 are not presented here. These nets can be found in [5].

guide our search. The logs show the allowed (positive) behavior, but they do not express the forbidden (negative) one.

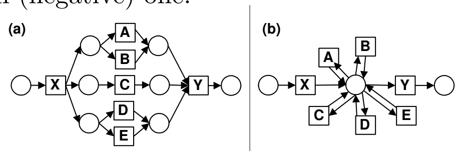


Fig. 3. Example of nets that can also reproduce the behavior for the log in Table 1. The problem here is that these nets allow for extra behavior that is not in the log.

A possible way to increase the probability that the GA will mine models that allow for none or little extra behavior is to use a hybrid version of evolutionary algorithm that uses heuristics in the search [3]. In our case, the hybrid version uses some heuristics to build the initial population. In short, the more often a task t is directly followed by a task t' (i.e. the subtrace "t, t'" appears in traces in the log), the higher the probability that individuals are built with a causality relation from t to t'. Details about the heuristics can be found in [5]. With this setting, the genetic algorithm could also find, most of the time, an individual that could parse all traces in the log. Furthermore, the genetic algorithm sometimes found an individual that is equal to the original net (see Table **??**). Note that the hybrid genetic algorithm performed better for nets with few or no parallelism. Additionally, the use of heuristics hindered the discovering of the non-free-choice construct because there is no direct relation between tasks $A - D$ and $B - E$, and if we remove the places $p3$ and $p5$ from the net in Figure 1, the resulting net can also parse all traces that the net in Figure 1 can parse.

Another possible solution is to improve the fitness function to make sure that the mined model is not only *complete* (parses all traces in a log) but it is also *minimal* (does not allow for classes of traces that cannot be derived from the log). The next section presents some ideas on how to do so.

5 Challenges and Some Ideas

Although our fitness measure F (see Subsection 3.2) punishes individuals with wrong AND/OR-split/join constructs and individuals with missing arcs, it does not punish individuals that allow for additional behavior not present in the log. Thus, our challenge is to include in the fitness function F a measure that punishes for extra behavior.

One possible solution to punish an individual that allows for undesirable behavior could be to build the *coverability graph* [6] of the mapped Petri net for this individual and check the fraction of event traces this individual can generate that are not in the log. The traces that express different paths of execution for parallelism are not considered as extra behavior. The main idea in this approach is to punish the individual for every extra event trace it generates. Unfortunately, building the coverability graph is not very practical and it is unrealistic to assume that all possible behavior is present in the log.

Another possibility is to check, for every marking, the *number of visible tasks that are enabled*. Individuals that allow for extra behavior tend to have more enabled tasks than individuals that do not. For instance, the nets in Figure 3 have more enabled tasks in most reachable markings than the net in Figure 1. The main idea in this approach is to punish more the individuals that have more enabled visible tasks during the parsing of the log.

6 Conclusions and Future Work

In this paper we presented a genetic algorithm to mine process models. The internal representation allows for the mining of process models that contain non-free-choice and invisible tasks. The current version can search the space defined by the set of tasks in the log and return a process model (individual) that can parse all event traces, regardless of how the initial population is built. This means that our genetic operators (crossover and mutation) are working as expected. However, the fitness measure needs to be improved to make sure that the mined models only express the behavior in the log. Our main challenge here is how to cope with the lack of negative examples. We do not have logs that show the forbidden (negative) behavior. Thus, the genetic algorithm has to work with what actually happened (positive examples), but it still should punish the individuals that allow for extra behavior that does not comply with the log. Some ideas to improve the fitness measure include (i) computing all the traces that an individual can produce (its coverability graph) or (ii) checking the amount of tasks that are enabled at every marking during the parsing of event traces. Our future work will focus on developing metrics to mine process models that are not only complete (express the behavior in the log), but are also minimal (do not allow for extra undesired behavior).

References

1. 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.
2. 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.
3. A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing.* Natural Computing. Springer-Verlag, Berlin, 2003.
4. 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.
5. A.K.A. de Medeiros, A.J.M.M. Weijters, and W.M.P. van der Aalst. Using Genetic Algorithms to Mine Process Models: Representation, Operators and Results. BETA Working Paper Series, WP 124, Eindhoven University of Technology, Eindhoven, 2004.
6. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.