A Genetic Algorithm for Process Discovery Guided by Completeness, Precision and Simplicity

Borja Vázquez-Barreiros, Manuel Mucientes, and Manuel Lama

Centro de Investigación en Tecnoloxías da Información (CiTIUS)
University of Santiago de Compostela, Spain
{borja.vazquez,manuel.mucientes,manuel.lama}@usc.es

Abstract. Several process discovery algorithms have been presented in the last years. These approaches look for complete, precise and simple models. Nevertheless, none of the current proposals obtains a good integration between the three objectives and, therefore, the mined models have differences with the real models. In this paper we present a genetic algorithm (ProDiGen) with a hierarchical fitness function that takes into account completeness, precision and simplicity. Moreover, ProDiGen uses crossover and mutation operators that focus the search on those parts of the model that generate errors during the processing of the log. The proposal has been validated with 21 different logs. Furthermore, we have compared our approach with two of the state of the art algorithms.

Keywords: Process mining, process discovery, Petri nets, genetic mining.

1 Introduction

In the last decade a great effort has been made for developing technologies to automate the execution of processes in different application domains such as industry, education or medicine [3]. In this context, a process is understood as a collection of tasks —or activities— with coordination requirements among them [8]. These tasks are performed by a set of actors to achieve the purpose of the process. Typically, these processes have a detailed description, i.e., there is a design of the process where its activities and the actors participating in these steps are clearly described. However, even in this situation there might be differences between what is actually happening and what is predefined in the process.

Based on this, Process Mining (PM) techniques are needed to get information about *what is really happening* in the execution of a process, and *not what the people think it is happening* [9]. Typically, these techniques use the log files that collect information about the events detected and stored by the information system in which the process has been executed. While PM techniques can be classified in different groups —*process discovery*, *conformance checking* or *enhancement*— this paper focuses its attention into the process discovery problem, i.e, the control-flow discovery, which aims to retrieve the process model that

S. Sadiq, P. Soffer, and H. Völzer (Eds.): BPM 2014, LNCS 8659, pp. 118–133, 2014.
© Springer International Publishing Switzerland 2014

represents the behavior recorded in an event log. These algorithms are used to discover the underlying process that has been followed by users to achieve an objective.

There has been a lot of work on process discovery [9,13,10,1,12,2,4]. Although some mining techniques use a specific target model for control flow discovery [4], most of the process discovery algorithms are based on Petri nets. These algorithms can be classified depending on the type of technique they applied. Thus *abstraction-based* algorithms [9,13], in general, retrieves simple models but with poor completeness. Other approaches, based on *heuristics* [12], although being robust to noise, do not guarantee optimal results in terms of completeness, as they focus on the main behavior of the log —also, they cannot handle all the common structures at once. Within the *search-based* algorithms, some techniques guarantee sound models [1] —not guaranteeing always the complete model as a solution—, and others can tackle all the different main behavior at once [2], but leaving simplicity aside. Other techniques, based on *theory of regions*, despite guaranteeing complete models [10], cannot handle noise and all the different pattern constructs. Summarizing, very valuable results have been achieved, but it is necessary to deep in the development of algorithms that guide its search towards complete, precise and simple models.

In this paper we present ProDiGen[1] (Process Discovery through a Genetic algorithm), a process discovery algorithm that guides its search towards complete, precise and simple models. The algorithm uses a hierarchical fitness function that takes into account completeness, precision and simplicity —with new definitions for both precision and simplicity— and uses heuristics to optimize the genetic operators: (i) a crossover operator that selects the crossover point from a Probability Density Function (PDF) generated from the errors of the mined model, and (ii) a mutation operator guided by the causal dependencies of the log. The proposal has been tested using 21 unbalanced logs, i.e, logs with many different traces and different frequencies. Furthermore, we have compared our approach with two of the state of the art process mining techniques, using a collection of conformance checking metrics.

The remainder of this paper is structured as follows. Sec. 2 presents the proposed genetic algorithm for process discovery. Sec. 3 shows the obtained results with the 21 logs and, finally, Sec. 4 points out the conclusions.

2 ProDiGen: Process Discovery through a Genetic Algorithm

The proposal of this paper (ProDiGen) is inspired by Genetic Miner [2], albeit there are several differences between them (Tab. 1). Although ProDiGen still codifies each individual[2] of the population[3] using the causal matrix representation [2], almost all of the mains steps of the genetic algorithm (GA) have been

[1] `http://tec.citius.usc.es/SoftLearn/ProDiGen.html`

[2] A candidate solution, i.e., a mined model.

[3] A collection of candidate solutions.

Table 1. Differences between ProDiGen and Genetic Miner

Fitness	The fitness is hierarchical and takes into account the completeness, precision and simplicity of the mined model.
Precision	Definition of a new method to measure the precision of a model.
Simplicity	Definition of a new method to measure the simplicity of a model.
Initialization	The solution of the Heuristics Miner is incorporated to the initial population as well.
Selection	Prodigen uses the binary tournament selection as selection mechanism.
Crossover	The crossover operator is guided by a Probability Density Function (PDF) generated from the errors of the mined model.
Mutation	The mutation operator is guided by the causal dependencies of the log.
Replacement	Steady-state approach, with a reinitialization criterium based on the improvement of the population.

Algorithm 1. ProDiGen

```
1  Initialize population
2  Evaluate population
3  t = 1, timesRun = initialTimesRun, restarts = 0
4  while t ≤ maxGenerations && restarts < maxRestarts do
5      Selection
6      Crossover
7      Mutation
8      Evaluate new individuals
9      Replace population
10     t = t + 1
11     if bestInd(t) == bestInd(t − 1) then
12         timesRun = timesRun − 1
13     if none of the individuals of the population have been replaced then
14         timesRun = timesRun − 1
15     if timesRun < 0 then
16         Reinitialize population
17         Evaluate population
18         timesRun = initialTimesRun, restarts = restarts + 1
```

modified. More specifically, one of the major changes takes place in the evaluation of the population, where completeness, precision and simplicity are considered in a hierarchical way. ProDiGen also defines (i) a new metric to measure the precision of each individual, and (ii) a new method to measure the simplicity of the model. Furthermore, we introduce heuristics to guide the genetic operators, focusing the search on those parts of the mined model that have errors and, also, reducing the search space to those models that are supported by the information in the log.

Algorithm 1 describes how ProDiGen works. The first three steps correspond to an initialization, where t represents the number of iterations, *timesRun* is used to detect situations in which the search gets stuck, and *restarts* counts the number of reinitializations. The evolution cycle of the algorithm starts at Alg. 1:4. This part will be repeated until the stopping criterion is fulfilled. The mains steps of the iterative part are the selection of the individuals, the crossover and mutation operations to generate new individuals, their evaluation, the replacement of the population, and the analysis of the population to detect blockages in the search process. All these steps are described in detail in the next sections.

2.1 Initialization

In ProDiGen, each individual codifies a workflow using a causal matrix representation [2]. A causal matrix can map any Petri net in terms of dependency relations —which tasks enable the execution of other tasks— as it represents the input and output dependencies of each activity of the model.

ProDiGen uses the same heuristics —based on the causality relations between tasks— described in [2] to generate the initial population. Moreover, we also add to the initial population the solution mined with the Heuristics Miner approach [12]. With this process, the dependency relations are captured using the Heuristics Miner and then, with ProDiGen, the different inputs and outputs bindings are optimized. We have empirically concluded that adding the Heuristics Miner solution to the initial population does not modify the model mined with ProDiGen. Nevertheless, the inclusion of this individual in the initial population *speeds up* the iteration at which the best individual is found: instead of relying only on randomly initialized individuals, ProDiGen also uses the dependency relations mined by Heuristics Miner.

2.2 Evaluation

The individuals of the population are evaluated taking into account completeness, precision and simplicity, combined in a hierarchical fitness function.

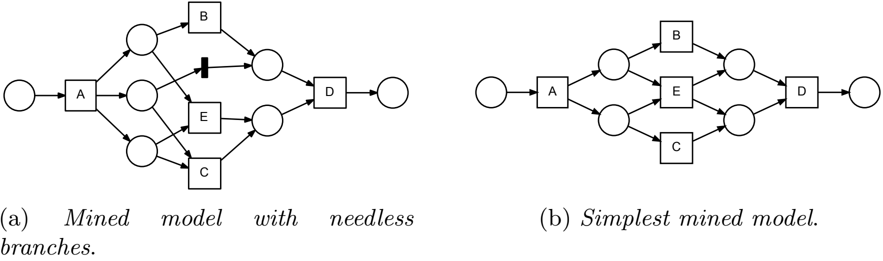
Completeness. We use the definition of completeness (C_f) described in [2], which takes into account the number of correctly parsed tasks [4], but also punishes the number of missing and not consumed tokens of the Petri net encoded in the individual —each missing or not consumed token represents a failure.

Precision. A model is precise when it reproduces the event traces of the log, not allowing for too much extra behavior, i.e, behavior that does not exist in the log. Our definition of precision considers all the activities that are enabled —tasks for which their input conditions are met when reproducing the log— while an individual parses the event traces of the log:

[4] If a task from an individual does not have the proper input arcs, that task will be incorrectly fired when reproducing the log, as its input conditions are not fulfilled.

$$P_f(L,\, CM) = \frac{1}{allEnabledActivities\,(L,\, CM)} \tag{1}$$

where *allEnabledActivities* is the sum of enabled activities after firing each activity of the log L by an individual CM. The idea behind this definition is to punish those individuals that enable too many activities during the parsing of the log, as they activate several paths that allow for extra behavior. Contrary to [2], ProDiGen does not consider the rest of the population in order to compute the precision of each individual, which can evolve regardless the precision of the rest of the population.



(a) *Mined model with needless branches.*

(b) *Simplest mined model.*

Fig. 1. Two possible solutions with the same completeness and precision

Simplicity. Completeness and precision give, by their own, a good indicator of how good is a mined model, but do not guarantee to find the simplest model. Hence, the third dimension of the fitness is simplicity. Although, there are several metrics that measure the complexity of a directed graph [6], there is no metric to measure the simplicity of a causal matrix. Instead of converting the causal matrix to a Petri net each time we want to measure the complexity of the model, we opted to define a new complexity metric for causal matrices. The new metric measures the complexity of a mined model based on the number of causal relations of an individual:

$$S_f(CM) = \frac{1}{\sum_{t \in CM} \left(\sum_{\Phi \in I(t)} |\Phi| + \sum_{\Psi \in O(t)} |\Psi| \right)} \tag{2}$$

where t is a task of the causal matrix CM, Φ is an element of the input function of t —I(t)—, and Ψ is an element of the output function of t —O(t)—. Therefore, the simplicity counts the number of causal relations of the model using the cardinality of the input and output subsets of the causal matrix.

To illustrate the relevance of simplicity to mine the correct model, lets assume a simple example with three different traces: $<< A, B, C, D >^3$, $< A, C, B, D >^2$, $< A, E, D >^4>$. Fig. 1 shows two mined models that have the same completeness and precision: (i) both can parse all the traces, i.e., $C_f = 1.0$; and (ii) they enable exactly the same number of tasks during the parsing (50), thus $P_f = 1/50$. However, the model in Fig. 1a has a $S_f = 1/16$, while the model in Fig. 1b has a $S_f = 1/14$ and, therefore, the second one is a better model[5].

Fitness. ProDiGen uses a hierarchical fitness function that establishes priorities among these three objectives:

$$F(a) > F(b) \iff \{C_f(a) > C_f(b)\} \vee \{C_f(a) = C_f(b) \wedge P_f(a) > P_f(b)\} \vee \{C_f(a) = C_f(b) \wedge P_f(a) = P_f(b) \wedge S_f(a) > S_f(b)\} \quad (3)$$

where $F(a)$, $C_f(a)$, $P_f(a)$ and $S_f(a)$, are respectively the *fitness*, *completeness*, *precision* and *simplicity* of a process model a. The advantage of using this hierarchical fitness function over a weighted fitness function is that, during the first stage of the evolutionary process, the GA focuses the search on those individuals that are complete. Once these individuals become representative in the population, the second level of the hierarchy takes the control, modifying the models that are complete in order to improve their precision. Finally, in the third stage, the fitness function guides the GA to improve the simplicity of those models that are both complete and precise.

2.3 Selection

ProdDiGen uses the binary tournament selection as selection mechanism. In a n-tournament selection, n individuals are randomly picked from the population —with replacement— and the best of them is selected. In this case, $n = 2$ —binary tournament selection.

2.4 Crossover

As the process models are represented through causal matrices, and the size of the causal matrix increases with the number of activities in the log, the number of possible crossover points could be really large —increasing significantly the search space. Thereby, we have noticed that picking the crossover point at random produces a poor performance of the crossover operator, as most of *the offspring have a fitness lower than their parents* after the crossover operation —the selected task of the individual to be crossed can be a correctly fired one.

[5] The difference between these two solutions —in terms of simplicity— is caused by the output function of the task A: the causal matrix of the model in Fig. 1a has $O(A) = \{\{B\,E\}, \{D\,C\}, \{C\,E\}\}$, which increases its complexity by 6. On the other hand, the causal matrix of the model in Fig. 1b has $O(A) = \{\{B\,E\}, \{C\,E\}\}$, increasing the complexity by 4.

ProDiGen makes the selection of the activity that is going to be crossed using a non uniform Probability Density Function (PDF). This PDF assigns a null probability of being selected to those activities that have been correctly fired during the parsing of the traces in the log. On the other hand, those activities that were incorrectly fired receive a uniform probability —inversely proportional to the number of incorrectly parsed activities— of being crossed.

Algorithm 2. Crossover operator

```
1  r ← getRandomNumber() ;        // returns a random number between [0,1)
2  if r < crossoverRate then
3      incorrectlyFiredActivities ← ∅
4      if fitness(parent_1) >= fitness(parent_2) then
5          incorrectlyFiredActivities ← set of incorrectly fired activities of parent_1
6      else
7          incorrectlyFiredActivities ← set of incorrectly fired activities of parent_2
8      if incorrectlyFiredActivities ≠ ∅ then
9          crossoverPoint ← randomly select an activity t from
           incorrectlyFiredActivities
10     else
11         crossoverPoint ← randomly select an activity t from the bag of all
           possible tasks in the log
12     offspring_1, offspring_2 ← doCrossover(parent_1, parent_2, crossoverPoint)
13     Repair offspring_1 and offspring_2
```

The selection of the crossover point is summarized in Alg. 2. By incorrectly fired activities we mean (i) activities that need extra tokens in their inputs to be fired, i.e, tasks that do not have the correct input arcs, and (ii) activities that have left tokens in their outputs after the parsing of the traces, i.e, tasks that do not have the correct output arcs. Therefore, during the evaluation process, the algorithm keeps track of the tasks with missing or extra tokens, and generates a bag of *incorrectlyFiredActivities* for each individual. Thereby, the crossover point is selected from the set of *incorrectlyFiredActivities* of the fittest parent (Alg. 2:4). Note that if the set of *incorrectlyFiredActivities* of the fittest individual is empty (Alg. 2:8), i.e, it has a completeness equal to 1, the crossover point is randomly chosen from the bag of all the possible tasks in the log (Alg. 2:11). After the crossover point is selected, the crossover is performed as defined in [2]. Thus, the crossover operator combines —by adding or merging subsets— the inputs for the selected task t of both parents, in order to generate the new inputs for t in the offspring. Finally this process is repeated for the output functions. As the input and output (I/O) functions of the crossover task can change by adding/removing causal dependencies, there may be inconsistencies between the I/O function of the crossover point and the rest of I/O functions of the individual — for example, a task t may have an output dependency with t', but t' does not

have the input dependency with t. Thereby, after each crossover, the individual has to be repaired (Alg. 2:13) to avoid these discrepancies between the input and output sets of the tasks. The repair process works as follows. For each task t' that was eliminated from O(t), the process checks if t' $\in$ O(t) —notice that t' can be in several subsets of O(t). If that is false, t has to be eliminated form I(t'). This process is repeated also for the input sets. On the other hand, when a task t' is added to O(t), the process checks if $t \in$ I(t'). If that is false, then t is added to I(t). A similar process is done for the inputs of t.

2.5 Mutation

The mutation operator modifies the population by (i) adding new material —new relations— to the individuals; (ii) removing causal relations; or (iii) reorganizing an input/output function, for instance, converting an AND-join into an OR-join.

Although ProDiGen uses the three mutation actions defined in [2], there are four major differences between our mutation operator and the one defined in Genetic Miner: (i) the individual is iteratively mutated until it is different from its parent —a mutation could generate an individual equal to its parent due to an useless mutation, for example, redistributing an empty set; (ii) only one task is affected by the mutation operator; (iii) individuals are always forced to mutate —the mutation probability is 1; and (iv) the task t' added to the I/O set of a task t must belong to the set of tasks that have an input/output dependency with t. The major goal of these modifications is to avoid duplicate individuals within the same population, or at least to minimize the duplicates. With these modifications, we have a more diverse population.

The mutation operator is summarized in Alg. 3. It uses two sets for the addition of a new task: *outputDependencies(t)* and *inputDependencies(t)*. ProDiGen uses these sets to reduce the set of tasks that are appropriate to be inserted in an I/O set, preventing the inclusion of a new task t' that never appears in a trace of t within the log. A first approach could be to include in the dependencies sets those tasks that have a dependency with t as calculated in the initialization phase. However, if we only take into account these dependencies, there will be not enough new material to discover all the different constructs. Therefore, *inputDependencies(t)* will be the set of tasks appearing before t in any trace of the log and, in the same way, *outputDependencies(t)* will be the set of activities that appear after t in any trace of the log. In this way, the mutation operator focuses only on those regions of the search space that represent information contained in the log. As a result, the success of the mutation operator increases, finding better offspring. Again, as the mutation operator can add or remove a task from an I/O set of a task t, there may be inconsistencies within the causal dependencies of the individual. Therefore, after each mutation the individual has to be repaired (Alg. 3:15), following the same strategy described in Sec 2.4.

Algorithm 3. Mutation operator

```
1  while the individual does not change do
2      Randomly choose one task t in the individual
3      mutationType ← getRandomNumber() ;    // returns a random number
       between [0,1)
4      if mutationType < 1/3 then
5          Randomly select a task t' from inputDependencies(t)
6          if getRandomNumber() < 1/2 then
7              Randomly choose one subset X ∈ I(t) and add the task t' to X
8          else
9              Create a new subset X, add the task t' to X, and add X to I(t)
10     else if mutationType < 2/3 then
11         Randomly choose one subset X ∈ I(t) and remove a task t' from X,
           where t' ∈ X. If X is empty after this operation, exclude X from I(t)
12     else
13         Randomly redistribute the elements from I(t)
14     Repeat from line 3, but using O(t) instead of I(t) and
       outputDependencies(t) instead of inputDependencies(t)
15     Repair the individual
```

2.6 Replacement

At each iteration, ProDiGen generates N offspring —being N the size of the population— as follows. Tournament selection randomly picks two parents from the current population. These individuals are modified by the genetic operators, creating two new individuals. This process is repeated until N offspring are generated. At this point, the parent population —current population— and the offspring population are joined and sorted —using the fitness. Finally the replacement operator selects the N best individuals. In order to maintain a diverse population, those repeated individuals are placed at the bottom of the ranking —keeping one representative in the original ranking position.

2.7 Reinitialization

A reinitialization takes place when the value of *timesRun* goes under 0 (Alg. 1:15), which indicates that the search process was not improving in the last iterations. This situation is detected in two ways. The first one (Alg. 1:11) is when the new population of an iteration has no new individuals —in comparison with the initial population of that iteration. The second indicator (Alg. 1:13) is the fact that the best individual does not improve. Each time that one of these situations is detected, *timesRun* decreases. The initial population after a reinitialization is generated in the same way as in the initialization stage. Moreover, ProDiGen also includes in the new population a mutation of the

Table 2. Process models used in the experimentation

	Activity structures										Log content	
Model	#Tasks	Sequence	Choice	Parallelism	Length-One Loop	Length-Two Loop	Arbitrary Loop	Structured Loop	Invisible tasks	Unbalanced AND-join/split	#traces	#events
g2 [2]	22	✓	✓	✓		✓	✓		✓		300	4501
g3 [2]	29	✓	✓	✓			✓	✓	✓		300	14599
g4 [2]	29	✓	✓	✓		✓				✓	300	5975
g5 [2]	20	✓	✓	✓				✓	✓		300	6172
g6 [2]	23	✓	✓	✓		✓			✓		300	5419
g7 [2]	29	✓	✓	✓			✓		✓		300	14451
g8 [2]	30	✓	✓	✓		✓	✓		✓	✓	300	5133
g9 [2]	26	✓	✓	✓		✓	✓		✓		300	5679
g10 [2]	23	✓	✓	✓				✓	✓		300	4117
g12 [2]	26	✓	✓	✓		✓		✓	✓		300	4841
g13 [2]	22	✓	✓	✓	✓	✓			✓	✓	300	5007
g14 [2]	24	✓	✓	✓			✓		✓	✓	300	11340
g15 [2]	25	✓	✓		✓	✓			✓		300	3978
g19 [2]	23	✓	✓	✓		✓			✓	✓	300	4107
g20 [2]	21	✓	✓		✓	✓		✓	✓		300	6193
g21 [2]	22	✓	✓					✓	✓		300	3882
g22 [2]	24	✓	✓	✓			✓		✓	✓	300	3095
g23 [2]	25	✓	✓	✓		✓				✓	300	9654
g24 [2]	21	✓	✓	✓				✓	✓	✓	300	4130
g25 [2]	20	✓	✓	✓	✓				✓		300	6312
EMT [1]	7	✓	✓	✓					✓		100	790

best individual of the last iteration. The maximum number of reinitializations is limited, and when it reaches the threshold (*maxRestarts*) ProDiGen ends.

3 Experimentation

This section describes (i) the validation of ProDiGen with 21 different logs using several conformance checking metrics, and (ii) the comparison of ProDiGen performance with two well-known state of the art process mining techniques: Heuristics Miner [12] and Genetic Miner [2].

3.1 Logs

ProDiGen has been validated with 21 different logs from [2] and [1]. Tab. 2 summarizes the structural complexity of these models ranging from 7 to 30

tasks. Some of the models used in the experimentation contain unbalanced AND-split/join points, i.e, there is not a one-to-one relation between the AND-split points and the AND-join points. Moreover, all the logs are imbalanced, i.e., they contain traces with very different frequencies. Thereby, with this experiment we can check whether the algorithm overfits or underfits the data due to the unbalanced frequencies of the traces in the log.

3.2 Metrics

The performance of ProDiGen over the different logs has been measured with two different sets of metrics: (i) metrics based on the original model; and (ii) metrics based on the event log.

Metrics Based on the Original Model. To compare the original and the mined models, we use the metrics defined in [2]:

- To quantify the *behavior similarity* between the original model and the mined one we use the metrics *Behavioral precision* (B_p) and *Behavioral recall* (B_r), which detect, respectively, if the mined model can process traces that cannot be parsed by the original model, and if the original model can parse traces that cannot be processed in the mined model. The mined model is as precise as the original one if $B_p = 1$ and $B_r = 1$: the closer the values of B_p and B_r to 1, the higher the similarity between the original and the mined models.
- On the other hand, to *measure the similarity from the structural point of view* of the mined model with respect to the original one, we use the metrics *Structural precision* (S_p) and *Structural recall* (S_r). They check, respectively, if there are causality relations of the mined model that are not defined in the original model, and if there are causality relations of the original model that are not defined in the mined model. When the original model has connections that do not appear in the mined model, S_r will take a value smaller than 1, and, in the same way, when the mined model has connections that do not appear in the original model, S_p will take a value lower than 1.

Metrics Based on the Log. Additionally to the four previously described metrics, we also use three metrics that do not require the original model as input:

- To measure the completeness we use the proper completion measure [5], which is the fraction of properly completed process instances. *Proper completion* (C) takes a value of 1 if the mined model can process all the traces without having missing tokens or tokens left behind.
- The precision is evaluated with the *alignment precision* (P) defined in [7], which, takes a value of 1 if all the behavior allowed by the model is observed in the log.
- Finally, for the simplicity (S) we use:

$$S = \frac{1}{1 + S'} \tag{4}$$

where S' is the *weighted P/T average arc degree* defined in [6]. The higher the value of S, the higher the simplicity. To measure these three metrics we used the tool CoBeFra [11].

Table 3. Results for the 21 logs

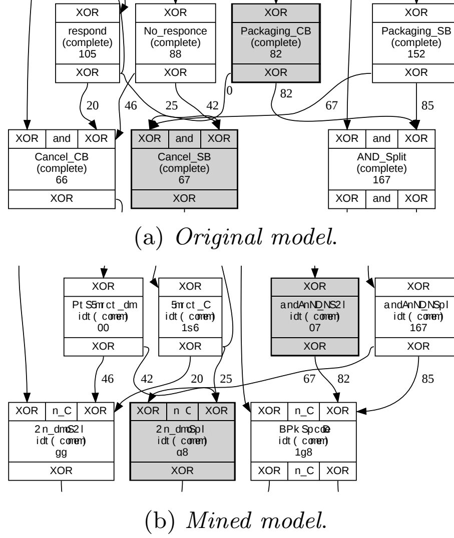
			g2	g3	g4	g5	g6	g7	g8	g9	g10	g12	g13	g14	g15	g19	g20	g21	g22	g23	g24	g25	EMT
ProDiGen	Model metrics	B_p	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.96	1.0
		B_r	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.99	1.0
		S_p	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.91	1.0
		S_r	1.0	1.0	0.97	1.0	1.0	1.0	0.94	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.98	0.91	1.0
	Log metrics	C	1.0	1.0	0.78	1.0	1.0	1.0	0.52	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.98	1.0
		P	0.9	0.82	0.98	0.98	0.95	0.88	0.86	0.92	0.89	0.97	0.93	0.93	0.86	0.92	0.78	0.91	0.9	0.58	0.89	0.74	0.87
		S	0.3	0.3	0.31	0.31	0.31	0.32	0.28	0.31	0.3	0.31	0.3	0.31	0.25	0.3	0.29	0.31	0.3	0.3	0.29	0.31	0.27
GM	Model metrics	B_p	1.0	0.61	0.78	1.0	1.0	1.0	0.84	0.96	0.99	1.0	0.98	0.61	0.8	0.98	1.0	1.0	0.97	0.57	0.83	0.81	1.0
		B_r	1.0	0.97	0.97	1.0	1.0	1.0	1.0	1.0	0.97	1.0	0.99	1.0	0.97	0.9	1.0	1.0	1.0	0.88	0.88	0.96	0.83
		S_p	1.0	0.81	0.81	1.0	1.0	1.0	1.0	0.97	0.9	1.0	0.95	0.95	0.88	0.95	1.0	1.0	0.85	0.76	0.75	0.76	0.85
		S_r	1.0	0.81	0.81	1.0	1.0	1.0	0.94	0.98	0.92	1.0	0.94	0.94	0.87	0.89	1.0	1.0	0.85	0.74	0.75	0.74	0.85
	Log metrics	C	1.0	0.31	0.59	1.0	1.0	1.0	0.26	0.48	0.48	1.0	0.75	1.0	0.15	0.2	1.0	1.0	0.43	0.2	0.72	0.41	0.3
		P	0.9	0.42	0.98	0.98	0.95	0.88	0.0	0.94	0.91	0.97	0.96	0.74	0.0	0.0	0.78	0.91	0.86	0.0	0.88	0.49	0.81
		S	0.3	0.31	0.3	0.31	0.31	0.32	0.26	0.3	0.29	0.31	0.3	0.31	0.24	0.29	0.29	0.31	0.3	0.28	0.3	0.28	0.3
HM	Model metrics	B_p	1.0	1.0	0.94	1.0	0.9	0.97	0.87	1.0	0.96	1.0	1.0	0.97	0.96	0.97	1.0	1.0	0.99	0.6	0.92	0.76	0.81
		B_r	1.0	0.98	0.92	1.0	0.98	0.97	0.99	0.98	0.95	1.0	1.0	0.97	0.98	1.0	1.0	1.0	0.99	1.0	0.88	0.94	0.96
		S_p	1.0	0.97	0.96	1.0	0.93	0.97	0.95	1.0	0.96	1.0	1.0	0.96	1.0	1.0	1.0	1.0	0.97	0.91	0.89	0.85	0.76
		S_r	1.0	0.97	0.86	1.0	0.97	1.0	0.86	1.0	0.96	1.0	1.0	0.92	0.86	0.9	1.0	1.0	0.91	0.94	0.81	0.85	0.74
	Log metrics	C	1.0	1.0	0.78	1.0	0.66	1.0	0.52	0.74	0.78	1.0	1.0	0.91	0.87	0.85	1.0	1.0	0.9	0.0	0.93	0.23	0.37
		P	0.9	0.83	0.99	0.98	0.93	0.9	0.86	0.93	0.9	0.97	0.93	0.92	0.87	0.93	0.78	0.91	0.9	0.0	0.86	0.71	0.85
		S	0.3	0.3	0.32	0.31	0.31	0.31	0.28	0.31	0.3	0.31	0.3	0.32	0.26	0.3	0.29	0.31	0.3	0.29	0.29	0.3	0.29

3.3 Results

Within this scenario, we have conducted an experimentation comparing ProDiGen with two of the state of the art most popular algorithms: Genetic Miner [2] and Heuristics Miner [12].

The values that have been used for the parameters of ProDiGen are: *maxGenerations* = 1,000, *initialTimesRun* = 35, population size = 100, crossover probability = 0.8 and *maxRestarts* = 5. For the Genetic Miner (GM), we selected the parameters indicated by the authors in [2]: *maxGenerations* = 5,000, population size = 10, crossover probability = 0.8, mutation probability 0.2, elitism rate = 0.2, selection type = tournament 5. For the Heuristics Miner (HM), we used the default parameters established in ProM 6.3 with the option *mine long distance dependencies* enabled.

Table 3 shows the results on the 21 logs. ProDiGen mines the same model as the original model in 17 of the logs —the values of the four model metrics are 1— while in the other four logs the mined model is very similar to the correct one. The difficulties in these 4 logs arise when (i) mining logs with parallel constructs with more than two branches and with two or more tasks in each branch, and (ii) when mining logs that came from models with unbalanced AND-join/split points. These type of patterns are even more difficult to mine considering that not all the possible combinations admitted by the original model are represented in the log, and not all the traces have the same frequency. Therefore, ProDiGen tries to better fit the most frequent behavior of the log, overfitting the data



(a) *Original model.*

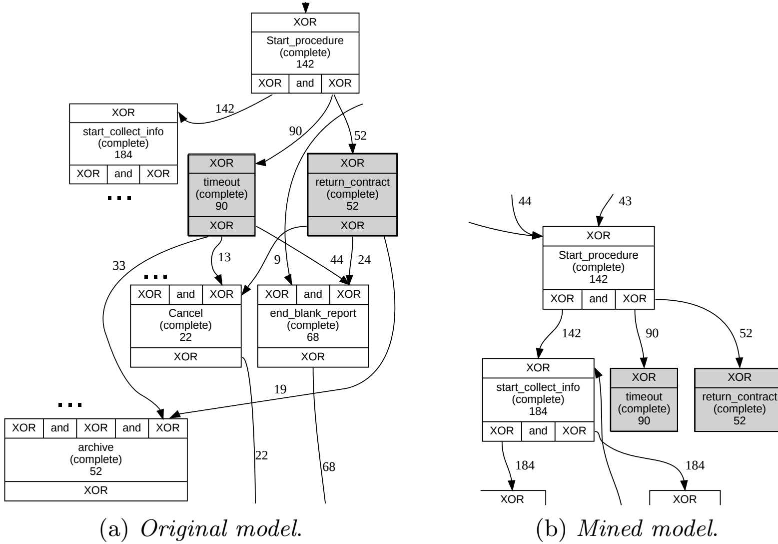
(b) *Mined model.*

Fig. 2. Detail of the original and mined models for log $g24$

when dealing with these constructs. We now discuss the details of the models incorrectly mined by ProDiGen:

– The results for log $g24$ (Fig. 2) show that the mined model is almost equal to the original one, except the only one relation between two tasks (tasks in grey, Fig. 2b). If we process the log with the original model (Fig. 2b) we can check that the missing relation is never used and, hence, it is impossible to mine that relation with the information of the log.
– The mined model for log $g8$ (Fig. 3) has a behavioral precision and recall equal to 1, i.e., the mined model can parse all the traces from the log and allows the same behavior as the original one with respect to the information contained in the log. However, the model is not complete because it cannot tackle the output dependencies of the tasks *timeout* and *return-contract*, considering them as final tasks —Fig. 3a shows the original output dependencies of these tasks. This results in a incomplete mined model, because all the traces involving these two tasks will have an extra token at the end of the parsing. The main problem with this log is that these two tasks are involved in the unbalanced AND-join/split, which cannot be correctly mined by ProDiGen.
– For log $g25$ the behavioral recall and precision are close to 1. This means that, even when the model is not as precise as the original, it does not allow for more extra behavior than the original one with respect to the log. Despite this model does not have an unbalanced AND-join/split point, it has many interleaving situations, which make very difficult to properly mine the correct relations of the different branches of the parallel construct.

– The mined model for log $g4$ has again a behavioral precision and recall of 1, showing that it expresses the same behavior as the original model with respect to the log. The main problem when mining this log is that ProDiGen cannot find the complete model because it discovers an extra final task due to the unbalanced AND-join/split point —the same problem as in log $g8$.



(a) *Original model.* (b) *Mined model.*

Fig. 3. Detail of the original and mined models for log $g8$

Table 3 also shows the results of the other algorithms. The main problem of GM is that it finds solutions with too many silent transitions[6] generating models with low precision and simplicity. On the other hand, HM focuses its search on the main behavior of the log —finding solutions with high levels of simplicity. Hence, it cannot find the original model on those logs that came from models with many interleaving situations, as it tries to better fit the most frequent behavior recorded in the log —as the logs are unbalanced, *not all the possible relations* have the same frequency.

Comparing the results of the three algorithms: ProDiGen correctly mines, i.e, finds the original model, the 81% (17 out of 21) of the cases; GM finds the original model in the 33% (7 out of 21) of the logs; and HM finds the original model in the 28% (6 out of 21) of the logs. Moreover, Table 3 also shows information about which algorithm retrieves better results for each metric —highlighted in grey. On those logs where ProDiGen did not find the original model —logs $g4$, $g8$, $g24$ and $g25$— it still obtains the best solution of the three algorithms.

[6] A silent transition is a type of activity used for routing purposes only, as it does not correspond to any activity in the log.

Based on this experimentation, we can conclude that using a hierarchical fitness function based on completeness, precision and simplicity, shows a great performance when mining unbalanced logs. Moreover, the inclusion of heuristics in the genetic operators also improves the results, as ProDiGen focuses the search over those regions that represent the behavior of the log.

4 Conclusions

We have presented ProDiGen, a genetic algorithm for process mining that can tackle all the different constructs at once, and obtains models that are complete, precise, and simple, while being robust to infrequent behavior and unbalanced logs. ProDiGen uses a new hierarchical fitness function that includes new definitions for precision and simplicity. Moreover, the proposal uses genetic operators that focus the search on specific parts of the model: (i) the crossover operator selects the crossover point based on the errors of the mined model; and (ii) the mutation operator is guided by the causal dependencies of the log. ProDiGen has been validated with 21 different models with all kind of workflow patterns and unbalanced logs. Results conclude that ProDiGen mine in most of the cases the original model, or a very similar, simple, and precise model that represents almost all the behavior of the log. Furthermore, ProDiGen has been compared with two of the state of the art algorithms, showing a better performance, and finding models that are complete, precise and simple.

Acknowledgment. This work was supported by the Spanish Ministry of Economy and Competitiveness under the project TIN2011-22935 and by the European Regional Development Fund (ERDF/FEDER) under the project CN2012/151 of the Galician Ministry of Education.

References

1. Buijs, J., van Dongen, B., van der Aalst, W.M.P.: Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity. International Journal of Cooperative Information Systems 23(01) (2014)
2. de Medeiros, A.: Genetic Process Mining. PhD thesis, Technische Universiteit Eindhoven (2006)
3. Dumas, M., ter Hofstede, A., van der Aalst, W.M.P.: Process-aware information systems: bridging people and software through process technology. Wiley-Interscience (2005)
4. Günther, C.W., van der Aalst, W.M.P.: Fuzzy mining – adaptive process simplification based on multi-perspective metrics. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 328–343. Springer, Heidelberg (2007)
5. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Information Systems 33(1), 64–95 (2008)
6. Sánchez-González, L., García, F., Mendling, J., Ruiz, F., Piattini, M.: Prediction of business process model quality based on structural metrics. In: Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.) ER 2010. LNCS, vol. 6412, pp. 458–463. Springer, Heidelberg (2010)

7. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.: Replaying history on process models for conformance checking and performance analysis. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 2(2), 182–192 (2012)
8. van der Aalst, W.M.P., Ter Hofstede, A.H., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases 14(1), 5–51 (2003)
9. van der Aalst, W.M.P., Weijters, A., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1128–1142 (2004)
10. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process discovery using integer linear programming. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 368–387. Springer, Heidelberg (2008)
11. vanden Broucke, S., Weerdt, J.D., Vanthienen, J., Baesens, B.: A comprehensive benchmarking framework (CoBeFra) for conformance analysis between procedural process models and event logs in ProM. In: 2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM), pp. 254–261. IEEE (2013)
12. Weijters, A., van der Aalst, W.M.P., de Medeiros, A.: Process mining with the heuristics miner-algorithm. Technische Universiteit Eindhoven 166 (2006)
13. Wen, L., Wang, J., Sun, J.: Mining invisible tasks from event logs. In: Dong, G., Lin, X., Wang, W., Yang, Y., Yu, J.X. (eds.) APWeb/WAIM 2007. LNCS, vol. 4505, pp. 358–365. Springer, Heidelberg (2007)