Generation of Process Using Multi-Objective Genetic Algorithm
Yoann Laurent, Reda Bendraou, Marie-Pierre Gervais

To cite this version:

HAL Id: hal-01088189
https://hal.archives-ouvertes.fr/hal-01088189
Submitted on 27 Nov 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Generation of Process using Multi-Objective Genetic Algorithm

Yoann Laurent
LIP6
UPMC Paris Universitas
France
yoann.laurent@lip6.fr

Reda Bendraou
LIP6
UPMC Paris Universitas
France
reda.bendraou@lip6.fr

Marie-Pierre Gervais
LIP6
University of Paris Ouest
Nanterre
France
marie-pierre.gervais@lip6.fr

ABSTRACT
The growing complexity of processes whatever their kind (i.e. business, software, medical, military) stimulates the adoption of process execution, analysis and verification techniques. However, such techniques cannot be accurately validated as it is not possible to obtain numerous and realistic process models in order to stress test them. The small set of samples and “toy” models publically available in the literature is usually insufficient to conduct serious empirical studies and thus, to validate thoroughly work around process analysis and verification. In this paper, we face this problem by proposing a process model generator using a multi-objective genetic algorithm. The originality of our approach comes from the fact that process models are built through a sequence of high-level operations inspired by the way a process modeler could have actually performed to model a process. A working generator prototype has been implemented and shows that it is possible to quickly generate huge, syntactically sound and user-tailored process models.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging—
Data generators

General Terms
Algorithms, Design

Keywords
Process, Generator, UML Activity, Genetic Algorithm

1. INTRODUCTION
One frustrating situation that most scientists have to face frequently is the inability to validate their approaches because of a lack of realistic data and models. By realistic, we mean not only having models of a given size but models exposing specific properties and complex constructs that can be comparable to what a modeler could produce in real projects. The community of process modeling and analysis particularly suffers from such lack. Indeed, with the growing complexity of processes whatever their kind (i.e. business, software, medical, military), many approaches and tools were proposed to verify and to analyze them. However, some approaches cannot be accurately validated as it is not possible to obtain numerous and realistic process models in order to stress test them. The small set of samples and “toy” models publically available in the literature is usually insufficient to conduct serious empirical studies and thus, to validate thoroughly work around process analysis and verification. They are also often in textual or an inappropriate formalism and need to be converted to the input format expected by the verification tool.

One solution could be to promote initiatives in order to put in place open repositories of real world models. However, such initiatives face privacy issues and thus only few organizations accept to share their models (ex. Only 150 models where submitted to the Moogle repository). Additionally, there is no guaranty about the correctness of these models or if they hold interesting properties required for testing the validity of the verification approach.

In the literature many approaches were proposed to automatically generate models for testing purposes [9, 2, 6, 11]. All these approaches concerned the generation of models that related to structural concerns i.e., mainly class diagrams, instances of EMF-based meta-models [9] and to our knowledge none of them addressed behavioral models. Their main goal was to test the scalability of tools and approaches. These generators were used to produce huge models for testing for instance the scalability of consistency checking languages, model comparison approaches or the generation of test models.

Process verification approaches are more concerned about checking behavioral properties present in process models than of their size (ex. does the process ends one day? Would activity X be executed before the end of the process, etc.). That’s why having an approach that randomly generates a given number of activities, edges, and object flows is not enough. These approaches need realistic process models, holding some workflow patterns [13], with complex constructs such as loops, forks, conditional branches with validated guard, the whole combined in a consistent way as it was modeled.
by a process modeler. This is what the approach we propose in this paper aims to.

Our contribution comes in a form of a process model generator that uses a multi-objective genetic algorithm [3]. The originality of our approach comes from the fact that process models are built through a sequence of high-level operations inspired from the catalogue of process change patterns proposed in [15]. A working and scalable generator prototype has been implemented which shows that it is possible to quickly generate huge, syntactically sound and user-tailored process models.

2. GENERATING PROCESS MODEL

Before presenting our solution, we first introduce the genetic algorithm and finally, apply it in order to generate process models through a sequence of high-level operations.

2.1 Genetic Algorithm

Genetic Algorithm (GA) [3] are probabilistic search algorithms that use the principle of natural selection (based on Darwin’s theory of evolution) to evolve iteratively a set of solutions (called population) toward an optimum solution. A potential solution is called a chromosome. A chromosome is composed of multiple genes. A gene is a distinct component of a potential solution.

During each evolution, natural selection is applied to determine which solutions survive and which are discarded.

In order to proceed to the selection process, a so-called fitness function is required to be able to evaluate how “good” a solution relative to other potential solutions. The fitness function is responsible for performing this evaluation and returning a positive integer number, or “fitness value”, that reflects how optimal the solution is (e.g., the higher the number, the better the solution). The fitness values are then used in a process of natural selection to choose which potential solutions will continue on to the next generation, and which will die out. However, the natural selection process does not obviously choose the top $x$ number of solutions. The solutions are instead chosen statistically such there is more chance that a solution with a higher fitness value will be chosen, but it is not guaranteed. Indeed, a solution can be temporally weaker than the others, but may evolve in few generations into something even better than the previous “better” solutions.

To evolve the population into a new one, genetic operations are applied on the population such as: (i) reproduction, i.e. making a copy of a potential solution, (ii) crossover, i.e. swapping gene values between two potential solutions, simulating the “mating” of the two solutions and (iii) mutation, i.e. randomly altering the value of a gene in a potential solution.

The evolution continues until a fixed termination goal is reached such as time limit or sufficient fitness achieved.

Thus, the outline of a genetic algorithm corresponds to:

1. **Genesis**: Creation of an initial set (population) of $n$ candidate solutions (randomly or provided).
2. **Evaluation**: Evaluate each member of the population using some fitness function.
3. **Survival of the Fittest**: Select a number of members of the evaluated population, favouring those with higher fitness scores.
4. **Evolution**: Generate a new population using genetic operations.

5. **Iteration**: Repeat steps 2-4 until the termination condition is met.

2.2 Using GA for Process Generation

The purpose of using a GA to generate processes is to simulate the natural way a process modeler could have actually followed to model a process. However, there are generally different objectives behind each generation of a process. For instance, if the purpose of the generation needs to test the execution scalability of an approach, one needs to influence the generation towards massively parallel processes. We identify three main generation objectives: (i) the size of the generated process (e.g., the process will contain approximately 100 nodes), (ii) the number of each elements (e.g., the process will contain more than 3 ForkNode and less than 30 Action), and (iii) constraints to specify the static structure of the process (e.g., all the ForkNode will have more than 4 output edges). An example of an application using such objectives could be to generate a sample of processes containing some Workflow Patterns [16] (supported by UML Activity diagrams in our case) to test a specific verification approach.

The objectives are then the soft goals which need to be achieved as good as possible. Adding objectives on the generation goals implies more restriction to the set of possible solutions. Of course, these objectives are not mutually exclusive.

Figure 1 shows how the GA is used to generate processes and is explained in the following. A chromosome corresponds to a process while process elements (i.e., nodes and edges) represent its genes.

**Genesis**: To configure the initial population, some information is required. (i) The length of the population, which corresponds to the maximum number of possible process solutions. A higher value ensures to find more satisfying solutions at the cost of increasing the total computation time. A big population also ensures a better diversity in the generated processes. (ii) The initial process which will evolve through the evolution. By default, the process corresponds to a simple process with an InitialNode and an ActivityFinalNode (since all processes have a start and an end). However, it is possible to use a user-defined process for the initial population (the generation will be a derivation of the process). Then, the initial process is copied into all the initial populations.

**Evaluation**: The fitness function evaluate the given chromosome regarding the defined objectives. Assuming that $p$ is a process candidate, $C_i$ is the desired size of the generated process, $C_m$ the margin accepted on the size $C$, $C_e$ the set...
which contains the desired number of each element, \( C_e \), the set of syntactical constraints and \( W_0, W_c, W_r \) the weight according to each objective. Let \( W \) be the sum of all the weight. Let \( \text{hold}(\text{candidate}, \text{objective}) \) be a function which returns 1 if the objective holds on the candidate and 0 otherwise. Let \( \text{size}(\text{candidate}) \) be a function which returns the size of the process. Let \( \text{margin}(x) \) be a threshold function used for the margin size of the models. The \( \text{fitness}(\text{candidate}) \) function returns a real number between \([0, 1]\) that reflects how optimal the solution is (higher value means that the solution is better):

\[
\text{margin}(x) = \begin{cases} 
0 & \text{iff } x \leq C_m \\
1 & \text{iff } x > C_m 
\end{cases}
\]

\[
\text{fitness}(p) = \frac{1}{1 + \text{margin}(\text{size}(p) - C_s)} W_s + \sum_{c \in C_c} \frac{\text{hold}(p, c) \text{card}(C_c)}{\text{card}(C_c)} W_0 + \frac{\sum_{c \in C_c} \text{hold}(p, c) \text{card}(C_c)}{\text{card}(C_c)} W_r
\]  

Let \( \delta \) be the acceptance threshold a real number between \([0, 1]\). Let \( \text{fitenough}(\text{candidate}) \) be the function which returns a boolean determining if the solution is considered fit enough (i.e., if the objectives are met). Note that a lower value assigned to \( \delta \) implies more rigidity in order to consider a chromosome fit. Thus, \( p \) is considered fit enough if:

\[
\text{fitenough}(p) = 1 - \text{fitness}(p) < \delta
\]

Survival of the Fittest: Selection must favour fitter candidates over weaker candidates but there are no fixed rules, there is no one strategy that is best for all problems. We use the most common fitness-proportionate selection technique called Roulette Wheel Selection (RWS) [1]. Conceptually, each member of the population is allocated a section of an imaginary roulette wheel. A proportion of the wheel is assigned to each of the possible selections based on their fitness value. The wheel is then spun and the individual associated with the winning section is selected. The wheel is spun as many times as is necessary to select the full set of parents for the next generation. To ensure that good candidates are not lost during each generation, we use the principle of elitism. Elitism involves copying a proportion of the fittest candidates, unchanged, into the next generation. The candidates which meet the objectives (i.e., fit enough) are directly added to the "elite" population and stop to evolve.

Evolution: At each evolution, some genetic operations are applied on each solution of the population. We use only the principle of mutation operations. In our case, a mutation is the application of a given change pattern. The change patterns have been introduced by Weber et al. [15] and represent a set of 18 high-level process adaptations. The application of a change pattern transforms a process schema \( S \) into another process schema \( S' \). The most common change patterns are presented on the figure 3 and an example of application is visible on figure 2. Due to space restriction, it is not possible to explain the whole set of change patterns. In order to generate realistic processes, there is a need to specify a probability on the chance to apply a given change pattern on the candidate. Indeed, when a modeler builds a process, there is more chances that he performs the serialInsert than the conditionalInsert patterns. Thus, the evolution function needs to take into account a user-defined probability on each change pattern. For example, augmenting the probability of the parallelInsert will augment the chance to generate massively parallel processes while lowering it will build more sequential processes. A fine-tuned probability on each change pattern enables the generation of realistic processes. Thus, the mutation applies a relevant change pattern according to its associated probability to be chosen. The entire population evolves in parallel.

**Iteration:** The generation halts when there is no improvement in the overall fitness observed after \( x \) generations (stagnation termination) or when the desired number of fit solution is reached. In addition, a timed-out condition ensures that the algorithm does not run indefinitely. Usually genetic algorithm seeks to find one optimal solution while here we are looking for multiple solutions which meet the objectives. Using a huge population to generate a smaller set of solutions prevents the convergence towards an homogeneous set of solutions.

When the evolution stops, the algorithm sends back all the candidates which are fit enough. Thus, these candidates are the solutions, i.e. the generated processes.

### 2.3 Correctness of the Generated Processes

The notions of correctness for a process model concern two aspects: (i) to verify if the process is well-formed (i.e., syntactical correctness), and (ii) to determine in advance, whether the model exhibits certain desirable behaviors (i.e., behavioral correctness).

Concerning the verification of the syntactical correctness, it corresponds to check if the syntax of the model respects its metamodel and its associated constraints. This kind of verification is well supported by many tools and approaches [8] and such constraints are checked almost instantaneously [5]. However, in our case the construction of the process using only change pattern ensure its syntactical correctness [15] without performing such verification.

Formal notions such as soundness [14] define behavioral anomalies in process models. Some advanced process modeling tools implement verification methods based on these notions to automatically detect such anomalies [7, 12].

![Figure 2: Construction of a process in 6 steps using only Insert Process Fragment patterns](image)

![Figure 3: Overview of the Change Pattern, taken from [15]](image)
The problem with behavioral correctness is that unless the whole state space is explored, it is not possible to provide evidence for it. Unfortunately, it is not possible to afford it at each step of the process generation since exploring the entire state space is notoriously an exponential problem which fails in computation time for huge models [7].

Figure 5 shows the application of the addControlDependency change pattern which implies the creation of a deadlock on the process. Both Action after the ForkNode require a token on all their input ControlFlow to start which is not possible with these two added ControlFlow. However, our goal while generating processes is to simulate how the process modeler builds processes. Therefore, the generated processes may contain behavioral anomalies the same way as a modeler can build a process with behavioral anomalies. Moreover, generating process with behavioral anomalies is an important point in order to test formal verification approaches.

3. EVALUATION

The prototype we developed is currently provided as an Eclipse Juno EMF plugin. We use the Watchmaker Framework [4] to implement the multi-objectives genetic algorithm. This framework provide an extensible, high-performance (multithreaded), object-oriented API to implement evolutionary and genetic algorithm in Java.

Figure 4 shows a screenshot of the prototype. The intent of this prototype is to assist the modeler by automatically generates UML 2.0 Activity-based processes in the form of XMI Instance. The generation is customizable in multiple ways:

- (label 1) Destination folder of the generation, number of nodes with margin and population used for the GA.
- (label 2) Probability of each change pattern.
- (label 4) Set a specific process to populate the initial population.
- (label 5) Number of each elements (equal and less/more than a value).

Figure 4: Prototype of the Process Generator integrated into Eclipse

Figure 5: Deadlock due to the application of the addControlDependency change pattern.

The problem with behavioral correctness is that unless the whole state space is explored, it is not possible to provide evidence for it. Unfortunately, it is not possible to afford it at each step of the process generation since exploring the entire state space is notoriously an exponential problem which fails in computation time for huge models [7].

Figure 5 shows the application of the addControlDependency change pattern which implies the creation of a deadlock on the process. Both Action after the ForkNode require a token on all their input ControlFlow to start which is not possible with these two added ControlFlow. However, our goal while generating processes is to simulate how the process modeler builds processes. Therefore, the generated processes may contain behavioral anomalies the same way as a modeler can build a process with behavioral anomalies. Moreover, generating process with behavioral anomalies is an important point in order to test formal verification approaches.

3. EVALUATION

The prototype we developed is currently provided as an Eclipse Juno EMF plugin. We use the Watchmaker Framework [4] to implement the multi-objectives genetic algorithm. This framework provide an extensible, high-performance (multithreaded), object-oriented API to implement evolutionary and genetic algorithm in Java.

Figure 4 shows a screenshot of the prototype. The intent of this prototype is to assist the modeler by automatically generates UML 2.0 Activity-based processes in the form of XMI Instance. The generation is customizable in multiple ways:

- (label 1) Destination folder of the generation, number of nodes with margin and population used for the GA.
- (label 2) Probability of each change pattern.
- (label 4) Set a specific process to populate the initial population.
- (label 5) Number of each elements (equal and less/more than a value).

Table 1: Generation step and building time to generate 100 processes using a population of 1000

<table>
<thead>
<tr>
<th>Size (10% margin)</th>
<th>Generation step</th>
<th>Building time</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>36</td>
<td>1 164ms</td>
</tr>
<tr>
<td>100</td>
<td>73</td>
<td>2 247ms</td>
</tr>
<tr>
<td>200</td>
<td>156</td>
<td>5 002ms</td>
</tr>
<tr>
<td>500</td>
<td>319</td>
<td>12 207ms</td>
</tr>
<tr>
<td>600</td>
<td>483</td>
<td>20 112ms</td>
</tr>
<tr>
<td>800</td>
<td>649</td>
<td>31 238ms</td>
</tr>
<tr>
<td>1000</td>
<td>816</td>
<td>45 040ms</td>
</tr>
</tbody>
</table>

Table 1: Generation step and building time to generate 100 processes using a population of 1000

Default value exists for each input configuration, the user only needs to specify a destination folder to start its first generation. For ease of use, the weight value of each objective and the δ value is already predefined such as \( W_s = 2, W_e = 1, W_c = 1 \), and \( \delta = 0.1 \).

All the following executions are done on a MacBook Air 2011 with the Intel Core i5 processor and 4 GB of RAM. Each result corresponds to the average timing of 100 executions.

We initialize the objectives (for the fitness function) such as \( C_s = 50 \) (number of nodes), \( C_m = 10\% \) (margin for the size), \( C_e = \{ Activity \rightarrow 1 \} \) (objective number of each element), \( C_c = \emptyset \) (no OCL constraints).

Concerning the parameter for the evolution process, let \( E_P \) be the set which associates to each change pattern a probability, \( E_s \) the size of the population and \( E_t \) the initial process. We initialize these parameter such as \( E_P = \{ \text{serialInsert} \rightarrow 10, \text{conditionalInsert} \rightarrow 1, \text{parallelInsert} \rightarrow 1, \text{delete} \rightarrow 1, \text{copy} \rightarrow 2 \} \) (probability on each change pattern), \( E_s = 100 \) (size of the population) and \( E_t \) uses the default process with a simple initial and final node.

Using these parameters, the average timing for generating one model which fulfill the defined objectives takes 74ms and needs 32 generation steps. An example of such generated model is visible on the label 6 of figure 4.

To test the scalability of the generation, we change the size of the population such as \( E_s = 1000 \) and run the evolution until 100 candidates are fit enough. Table 1 shows the average generation step and building time needed to generate the processes regarding a specific size \( (C_s) \). The experimentation shows that the approach is able to fastly generate huge and realistic processes such as the building time is linear with the size of the generated process.

4. RELATED WORK

Mougenot et al. [9] propose a uniform random generator
of huge metamodel instances. The approach relies on the Boltzmann random sampling method that generates, in a scalable way, uniform samplings of any given size. In addition, the approach is able to influence the generation output by adding ponderations on elements. However, this approach does not support the additional constraints on the syntax and produces models only valid to its metamodel.

Brottier et al. [2] present a formalism to generate random constrained models, which is used in the context of model transformation testing. The approach consists in deriving a set of inputs example models to random alike instances using an homemade algorithm. One drawback of the approach is that it require instance of the model to generate others, therefore the outputted models may have a lot of similarities.

Ehrig et al. [6] present an algorithm that can generate instances of metamodels by transforming it into a set of graph specification rules. Then, the rules are selected randomly in order to perform the generation.

Pietsch et al. [11] present a generator of test models for model processing tools. They use a stochastic controller to apply low-level operations (create, delete, update, move) and more complex operations (composed of these low-level operations) to elements of the model. Elements are chosen using a random selection method inspired from the genetic algorithms one.

Unfortunately, [9, 6, 11, 2] are not adapted for the generation of behavioral models (e.g., process) and may produce unrealistic processes since the possibility of influencing the generation (add some ponderations) is either not available or handled only on given elements/attributes. The problem comes from the fact that these approaches aim at generating static models and are not tailored towards behavioral models generation.

5. CONCLUSION AND FUTURE WORK

This paper presented a multi-objective genetic algorithm to generate processes. The resulting generator has three interesting particularities. First it is scalable, the complexity of the generating algorithm is linear with the size of the generated processes. The size is controllable and allows to generate quickly huge processes. It also generates processes with multi-objectives allowing to generate user-tailored process models. Finally, the generation ensures syntax correctness through the sequence of change pattern and simulates the way a process modeler could have actually done to model a process. This ensure realistic processes while simulating the errors a modeler may have done.

The algorithm can be easily extended with new generation objectives by modifying the fitness function. The objectives presented on this paper focus on the syntactical aspects with associated constraint on it. However, we can imagine behavioral objectives by adding a process engine or a model-checker inside the fitness function (e.g., the process can only have 2 Action simultaneously executing). Moreover, some new genetic operators based on the crossover principle might improving the efficiency by combining multiple candidates (or part of the candidate) in order to converge faster to the objectives.

The generation focus only on the structural aspect of the process (i.e., the workflow). A possible extension might be to generate also the organizational information (such as resources, actors, deadline...) associated to the workflow. However, these information are generally domain-dependent and it can be hard to find a generic solution that suits all kinds. Moreover, one drawback of the set of change pattern comes from the fact that they focus only on the control-flow. A set of change patterns including data elements must be used to generate the data-flow and the control-flow in a unified way.

Finally, the generation technique used here opens the way to broader applications than generating process samples. For instance, by initializing the population with a given process and setting the right goal into the fitness function, it might be possible to automatically search for a derivation of the process which meet the desired needs (e.g., towards automatic correction of behavioral errors).

6. ACKNOWLEDGMENTS

The authors’ work is funded by the MERgE project (ITEA 2 Call 6 11011).

7. REFERENCES