PLG: a Framework for the Generation of Business Process Models and their Execution Logs

Andrea Burattin and Alessandro Sperduti

Department of Pure and Applied Mathematics
University of Padua, Italy
{burattin,sperduti}@math.unipd.it

Abstract. Evaluating process mining algorithms would require the availability of a suite of real-world business processes and their execution logs, which hardly are available. In this paper we propose an approach for the random generation of business processes and their execution logs. The proposed approach is based on the generation of process descriptions via a stochastic context-free grammar whose definition is based on well-known process patterns. An algorithm for the generation of execution instances is also proposed. The implemented tools are publicly available.

Keywords: process mining; business processes; log generation; Petri net; benchmark dataset.

1 Introduction

Process mining aims to discover the structure and relations among activities starting from business process logs. An important issue concerning the design of process mining algorithms is their evaluation: how well the reconstructed process model matches the actual process? This evaluation requires the availability of an as-large-as-possible suite of business processes logs and the corresponding original models (necessary for the comparison with the mined ones). In Fig. 1 we give a visual representation of such evaluation "cycle".

Unfortunately, it is often the case that just few (partial) log files are available, while no clear definition of the business process that generated the log is available. This is because many companies (the owners of "real" processes and logs) are reluctant to make public their own private data. Of course, the lack of extended process mining benchmarks is a serious obstacle for the development of new and more effective process mining algorithms. A way around this problem is to try to generate "realistic" business process models together with their execution logs.

In this paper, we present a new tool, the "Processes Logs Generator" (or PLG), developed for the specific purpose of generating benchmarks. It allows to: *i)* generate a random (hopefully "realistic") business process (according to some specific user-defined parameters); *ii)* "execute" the generated process and register each executed activity.

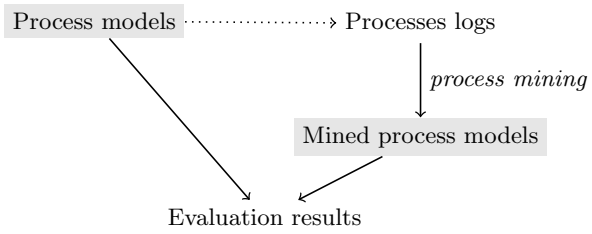

Fig. 1: The evaluation "cycle" for process mining algorithms.

The idea of generating process models for evaluating process mining algorithms is very recent. In [5][1] van Hee & Liu presented an approach to generate Petri nets representing processes. Specifically, they suggested to use a top-down approach, based on a stepwise refinement of Workflow nets [8], to generate all possible process models belonging to a particular class of Workflow networks (Jackson nets). A related approach is presented in [1], where the authors proposed to generate Petri nets according to a different set of refinement rules. In both cases, the proposed approaches do not address the problem of generating traces from the developed Petri nets.

2 The process generation phase

In this section the procedure for the generation of a business process is presented together with a description of the model we used.

Since our final aim is to ease the generation of business process models by the user, we decided to adopt a very general formalism for our process model description. Petri net [6] models are unambiguous and in-depth studied tools for process modelling, however controlling the generation of a complex process model via refinement of a Petri net may not be so easy for an inexperienced user. For this reason, we decided to model our processes via dependency graphs. A dependency graph is defined as a graph $G = (V,\ E,\ a_{start} \in V,\ a_{end} \in V)$ where V is the set of vertices and E is the set of edges. The two vertices a_{start} and a_{end} are used to represent the "start" and the "end" activities of the process model. Each vertex represents an activity of the process (with its possible attributes, such as author, duration, ...), while an edge $e \in E$ going from activity a_1 to a_2 represents a dependency relationship between the two activities.

In order to proceed with the presentation of our proposal, we need to introduce some definitions. Let's consider $v \in V$. The set of incoming activities for v is defined as $\text{in}(v) = \{v_i \mid (v_i, v) \in E\}$; its set of exiting (or outgoing) activities as $\text{out}(v) = \{v_i \mid (v, v_i) \in E\}$; the value of the fan-in of v is defined as $\overrightarrow{\deg}(v) = |\,\text{in}(v)|$ (i.e. the number of edges entering in v), while its fan-out is defined as $\overrightarrow{\deg}(v) = |\,\text{out}(v)|$ (i.e. the number of edges exiting from v).

[1] We discovered this work through the IEEE CIS Task Force on Process Mining.

In order to be able to correctly represent parallel execution (AND) and mutual exclusion (XOR) we introduce functions $\mathcal{T}_{out} : V \to \{\text{AND}, \text{XOR}\}$ and $\mathcal{T}_{in} : V \to \{\text{AND}, \text{XOR}\}$ which have the following meaning. For every vertex (i.e. activity) a with $\deg^{\rightarrow}(a) > 1$, $\mathcal{T}_{out}(a) = \text{AND}$ specifies that the flow has to jointly follow all the outgoing edges, while $\mathcal{T}_{out}(a) = \text{XOR}$ specifies that the flow has to follow only one of the outgoing edges. The meaning of $\mathcal{T}_{in}$ is analogous but it is referred to the type of the incoming edge (the type of join).

The strategy we adopt for the generation of a process is based on the recursive composition of basic patterns. The basic patterns we consider are (they correspond to the first patterns described in [7]): *i)* the direct succession of two workflows; *ii)* the execution of more workflows in parallel; *iii)* the mutual exclusion choice between some workflows; *iv)* the repetition of a workflows after another workflow has been executed (as for "preparing" the repetition).

The idea is to use these basic patterns for the generation of the process via a grammar whose productions are the patterns. Formally, we consider a context-free grammar $G_{Process} = \{V, \Sigma, R, P\}$, where $V = \{P, G, G', G_{\hookleftarrow\rho}, G_{\wedge}, G_{\otimes}, A\}$ is the set of non-terminal symbols, $\Sigma = \{;, (,), \hookleftarrow\rho, \wedge, \otimes, a_{start}, a_{end}, a, b, c, \dots\}$ is the set of all terminals (their "interpretation" is described in details in [3]), and R is the set of productions:

$$
\begin{array}{ll}
P \to a_{start} \ ; \ G \ ; \ a_{end} & \qquad G_{\hookleftarrow\rho} \to (G' \hookleftarrow\rho \ G) \\
G \to G' \mid G_{\hookleftarrow\rho} & \qquad G_{\wedge} \to G \wedge G \mid G \wedge G_{\wedge} \\
G' \to A \mid (G;G) \mid (A;G_{\wedge};A) \mid (A;G_{\otimes};A) & \qquad G_{\otimes} \to G \otimes G \mid G \otimes G_{\otimes} \\
A \to a \mid b \mid c \mid \dots &
\end{array}
$$

P is the starting symbol. Using the above grammar, a process is described by a string derived from P. It must contain a starting and a finishing activity and, in between, there is a sub-graph G. A sub-graph can be either a "single sub-graph" or a "repetition of a sub-graph". Let's start from the first case: a sub-graph G' can be a single activity A; the sequential execution of two sub-graphs $(G;G)$; or the execution of some activities in "AND" $(A;G_{\wedge};A)$ or "XOR" $(A;G_{\otimes};A)$ relation. It is important to note that the generation of parallel and mutual exclusion edges is "well structured", in the sense that there is always a "split activity" and a "join activity" that starts and ends the edges. The repetition of a sub-graph $(G' \hookleftarrow\rho \ G)$ is described as follows: each time we want to repeat the "main" sub-graph G', we have to perform another sub-graph G; the idea is that G (that can just be a single activity) corresponds to the "roll-back" activities required in order to prepare the system to repeat G'. The structure of $G_{\wedge}$ and $G_{\otimes}$ is simple and it expresses the parallel execution or the choice between at least 2 sub-graphs. Finally, A is the set of alphabetic identifiers for the activities (actually, this describes only the generation of the activity name, but the implemented tool "decorates" it with other attributes, such as a unique identifier, the originator, ...).

In order to allow the control on the complexity of the generated processes, we added a probability to each production. This addition required the introduction of user defined parameters to control the probability of occurrence into the generated process of a specific pattern. Besides that, for both the parallel pattern

and the mutual exclusion pattern, our framework requires the user to specify the maximum number of edges ($m_\wedge$ and $m_\otimes$) and the probability distribution that calculates the number of branches to be generated. The system will generate, for each AND-XOR split/join, a number of forks between 2 and $m_\wedge$ or $m_\otimes$, according to the given probability distribution.

In the current implementation, the system supports the following probability distributions: uniform distribution; standard normal (Gaussian) distribution and beta distribution (with α and β as parameters). These distributions generate values between 0 and 1 that are scaled into the correct interval ($2 \dots m_\wedge$ or $2 \dots m_\otimes$). The resulting values indicate the number of branches to be generated.

3 The execution of a process model

The procedure used to record the execution of the input activity and its successors (via a recursive invocation of the procedure) is reported in Algorithm 1. The two input parameters represent the current activity to be recorded and a stack containing stopping activities (i.e., activities for which the execution of the procedure has to stop), respectively. The last parameter is used when there is an AND split: an instance of the procedure is called for every edge but it must stop when the AND join is reached because, from there on, only one instance of the procedure can continue. The first time, this procedure is called with: ActivityTracer(a, $\emptyset$) where a is the starting activity of the process.

This algorithm is explained in details in [3]; it has to record the execution of an activity and then call itself on the following activity, considering all the possible cases ($\deg^{\rightarrow}(a) = 0$, $\deg^{\rightarrow}(a) = 1$ or $\deg^{\rightarrow}(a) > 1$). The function RecordActivity(a) is the one that writes the activity logs when executing the process; it can also introduce noise and information on the duration of the activity itself.

4 The implemented tool

The whole procedure has been implemented in a tool[2] developed in Java language. The implementation is formed by two main components: a library (PLGLib) with all the functions currently implemented and a visual tool, for the generation of one process. The idea is to have a library that can be easily imported into other projects and that can be used for the batch generation of processes. In order to have a deep control on the generated processes we added another parameter (with respect to the probabilities described in Section 2): the maximum "depth". With this, the user can control the maximum number of non-terminals to generate. Suppose the user sets it to the value d; once the grammar has nested d instances of G', then the only non-terminal that can be generated is A. With this parameter there is the possibility to limit the maximum "depth" of the final process.

[2] Available, at `http://www.processmining.it/sw/plg`.

Algorithm 1: for the execution of an activity and its successors.

ActivityTracer(a, s)

Input: a: the current activity
s: a stack (last-in-first-out queue) of activities

```
1  if s = ∅ or top(s) ≠ a then
2      RecordActivity(a)
3      if deg→(a) = 1 then
4          ActivityTracer(out(a), s)                          // recursive call
5      else if deg→(a) > 1 then
6          if T_out(a) = XOR then
7              a_1 ← random(out(a))                  // random outgoing activity
8              ActivityTracer(a_1, s)                         // recursive call
9          else if T_out(a) = AND then
10             a_j ← join(a)                        // join of the current split
11             push(s, a_j)
12             foreach a_i ∈ out(a) do
13                 ActivityTracer(a_i, s)                     // recursive call
14             end
15             pop(s)
16             ActivityTracer(a_j, s)                         // recursive call
17         end
18     end
19 end
```

The tool uses many libraries from ProM [4]. For storing the execution logs we use MXML. In the visual interface, we also implemented the calculation of two metrics for the new generated process, described in [2] (Extended Cardoso metric and the Extended cyclomatic one).

In Fig. 2 three screenshots of the GUI are shown. They give an idea of how the proposed tool allows to drive the creation of random processes, to configure all the parameters, and to visualize the obtained process as a Petri net.

5 Conclusions and future works

In this paper, we have proposed an approach for the generation of random business processes in order to ease the evaluation of process mining algorithms. The proposed approach is based on the generation of process descriptions via a (stochastic) context-free grammar whose definition is based on well-known process patterns; each production of this grammar is associated with a probability and the system generates the processes according to these values.

The work presented in this paper can be considered a first step to address the problem of random generation of business processes, and much more work has to be done before reaching a complete and satisfactory solution. Concerning

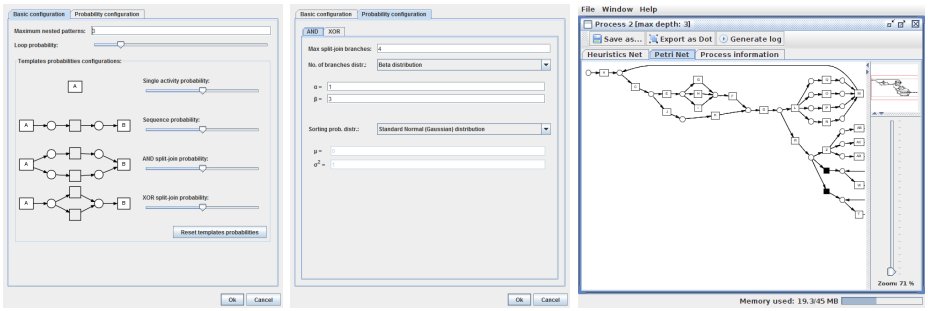

Fig. 2: Three screenshots of the implemented application. From left to right: two configuration panels and the process presentation window.

the generation of processes, the next goal to be achieved is the characterization of the space of the processes generated by our approach. Another open issue is on how much the generated processes can be considered "realistic": while using process patterns for their generation increases the probability to generate a realistic process, it would be nice to have control on this issue.

Acknowledgements

This work was supported by SIAV S.p.A. We thank Prof. Dr. Ir. W.M.P. van der Aalst, Prof. Dr. K.M. van Hee and Liu Zheng for their important suggestions.

References

1. Bergmann, G., Horváth, A., Ráth, I., Varró, D.: A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In: ICGT '08: Proceedings of the 4th international conference on Graph Transformations. pp. 396–410. No. i, Springer-Verlag, Berlin, Heidelberg (2008)
2. Bisgaard Lassen, K., van Der Aalst, W.M.P.: Complexity Metrics for Workflow Nets. Information and Software Technology 51(3), 610–626 (2009)
3. Burattin, A., Sperduti, A.: PLG: a Process Log Generator (2010), `http://www.processmining.it/publications`
4. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The ProM framework: A new era in process mining tool support. Application and Theory of Petri Nets 3536, 444–454 (2005)
5. van Hee, K.M., Liu, Z.: Generating Benchmarks by Random Stepwise Refinement of Petri Nets. In: Proceedings of workshop APNOC/SUMo (2010)
6. Peterson, J.L.: Petri Nets. ACM Computing Surveys (CSUR) 9(3), 223–252 (1977)
7. Russell, N., Ter Hofstede, A.H.M., van Der Aalst, W.M.P., Mulyar, N.: Workflow control-flow patterns: A revised view. BPM Center Report BPM-06-22, BPMcenter.org (2006)
8. van Der Aalst, W.M.P., van Hee, K.M.: Workflow management: models, methods, and systems. The MIT press (2004)