Generation of a Set of Event Logs with Noise

Ivan Shugurov
International Laboratory of
Process-Aware Information Systems
National Research University
Higher School of Economics
33 Kirpichnaya Str., Moscow, Russia
Email: shugurov94@gmail.com

Alexey A. Mitsyuk
International Laboratory of
Process-Aware Information Systems
National Research University
Higher School of Economics
33 Kirpichnaya Str., Moscow, Russia
Email: amitsyuk@hse.ru

***Abstract*—Process mining is a relatively new research area aiming to extract process models from event logs of real systems. A lot of new approaches and algorithms are developed in this field. Researches and developers usually have a need to test end evaluate the newly constructed algorithms. In this paper we propose a new approach for generation of event logs. It serves to facilitate the process of evaluation and testing. Presented approach allows to generate event logs, and sets of event logs to support a large scale testing in a more automated manner. Another feature of the approach is a generation of event logs with noise. This feature allows to simulate real-life system execution with inefficiencies, drawbacks, and crashes. In this work we also consider other existing approaches. Their forces and weaknesses are shown. The approach presented as well as the corresponding tool can be widely used in the research and development process.**

***Keywords*—Process mining, Petri net, event log, event log generation, ProM.**

I. Introduction

In this paper we present the approach for generation of a set of event logs. This work has been done within the bigger project related to a process mining research.

Process mining is a research area which aims to discover, monitor and improve real processes by extracting knowledge from event logs available in today's information systems [1], [2].

Two main fields of process mining are: process discovery and conformance checking. *Process discovery* [3] aims to solve the following problem: Given an event log consisting of a collection of traces, construct a Petri net that adequately describes the observed behaviour [1]. *Conformance checking* [4] aims to solve the problem as follows: Given an event log and a Petri net, diagnose the differences between the observed behaviour (i.e., traces in the event log) and the modelled behaviour [1].

Process models have applications in different fields of a modern industry. Banking, insurance, software engineering, and production management are examples of such fields.

Enormous work has been done for developing the process mining algorithms. ProM tool is a framework which gathers the majority of approach implementations for process mining [5], [6]. Core part of the ProM has been developed using Java over the last years by the process mining group at the Eindhoven University of Technology. This tool is open-source and it can be downloaded from the Internet.

ProM contains a wide variety of plug-ins. However researchers are continuously inventing new and more sophisticated methods for process mining. Every new method should be tested and evaluated in different ways. The first step of evaluation for every process mining method are tests using with artificial event logs. In this work we propose a new tool which allows to generate artificial event log with defined properties.

Researchers describe the incredible growth of data [7]. Big data is a new field of research which aims to process huge amounts of data in different industry sectors. One of the main challenges of modern process mining is to turn torrents of event data (Big Data) into valuable insights related to performance and compliance [8]. A lot of work being done now explores this direction.

In order to support these research we enrich capabilities of our tool to generate sets of event logs with defined properties. This is the first main feature of our method for log generation. Another feature serves to add noise. Real data often contains noise and inefficiencies which should be filtered (or processed) by an evaluated algorithm. Researchers have a need to evaluate new algorithms using event logs containing noise with special characteristics. In this paper we propose approach for noise adding in generated event logs.

All the ideas and approaches considered in this paper are implemented as a plug-in for ProM tool. We used standard data structures and approaches accepted in ProM community [5], [6]. Thus, our implementation can be easily used and integrated.

The remainder of this work is organized as follows. In section II we analyse other works in which log generation is considered. Section III gives a description of the tool, approaches and algorithms. Section IV concludes the paper.

II. Related Work

When creating new algorithms or improving already known in the young area of process mining it is crucial to have the possibility of multiple generation of process logs for a specific model. Developments in this area help researchers not only to verify concepts of algorithms but also to improve them based on model behaviour. When we provide a researcher an opportunity to manipulate a big number of behavioural examples of a model, it leads to higher quality of products being developed. Several tools have been developed to handle

this task. In this section we will take a look at existing tools for creation of event logs. We consider their main features, strengths and weaknesses.

a) CPN Tools (see [9]): CPN Tools is a widely used program to work with colored Petri nets.It supports the visual editing of Petri nets, simulation and analyses. This specific extension of CPN tools provides the possibility to generate random events log based on a given Petri net and produce the result log in MXML considering that the log will be used by ProM. CPN Tools has more or less usable GUI, but it is not intuitive. The main difficulty of a log creation is that it implies writing scripts in rarely used Standard ML language which leads to problems with extension of the tool and adapting it for a specific task. A user has to learn additional functional language. At the same time, the tool has a lot of applications in the field of colored Petri nets analysis and simulation.

b) Process Log Generator (see [10]): Process Log Generator (PLG) is a plug-in for ProM framework which enables to create random BPMN models from common workflow patterns and to simulate execution of these processes. PLG implements models customization by changing basic pattern percentages: loop percentage, single activity percentage, sequence percentage, AND split-join percentage, XOR split-join percentage. Furthermore, it gives users an opportunity to select distribution from Standard Normal, Beta and Uniform which is used to choose between random methods designated to decide which activity will be used. Noise log records can be generated throughout simulation of execution and it is possible to choose noise level. This tool is very useful for big scale brute force testing of an algorithm. The plug-in generates a set of models and an execution log for each model. Unfortunately, a user can not use existing model for logs generation. Thus, one can not make a fine adjustment of an experiment.

c) SecSy Tool (see [11]): Another instrument for a generation of event logs is SecSy tool. SecSy has been developed in a form of a standalone application allowing flexible settings of process models and their executions. It can create sets of logs per one run and add some deviations from the original model. The results can be produced in both MXML and XES formats. This tool is made to run experiments with security-oriented information systems. It allows to generate special event logs with particular parameters useful for security analysis of processes. Unfortunately, this orientation imposes restrictions on models which can be used by tool. Resulting event logs are also hardly useful when testing non-security-oriented algorithms.

d) Manual generation: Manual generation of logs has evident limitations and disadvantages including the necessity of learning XES or MXML standard. Creation of a big number of logs through manual generation is extremely tedious and inevitably leads to a tremendous quantity of mistakes. It is also very time-consuming activity even if a researcher has enough experience.

As we've seen, all of these tools have inconveniences. Using PLG user cannot use any existent models, because they are generated automatically. These tools do not provide possibility to change probabilities of outputs to be fired. None of them apart from CPN Tools do not support visual editing of models. When running some tools even a small mistake may cause significant deviation of results and give false view about correctness of algorithms.

III. Tool Overview

A. Functionality

In this paper we present the tool that intended to help researchers to generate sets of event logs by a Petri net replay. Petri net is a mathematical modelling language also known as a place/transition net. It is commonly used for modelling and representation of processes and systems. *Petri net* is a directed bipartite graph constructed from the following elements:

- *Transitions* signified by bars. They serve as events which may occur.
- *Places* signified by circles. They serve as conditions and connectivity elements.
- Directed *arcs* signified by arrows.

Out tool uses a Petri net as a model for event logs generation. We use this modelling language because of its prevalence among researchers who work on business process management [1]. Formally, an event log is a multiset of traces, where each trace is a sequence of events describing the life-cycle of a particular process instance [1]. Each event is a record representing some activity of a system (or model of a system). In figure 1 the example of a Petri net is shown.

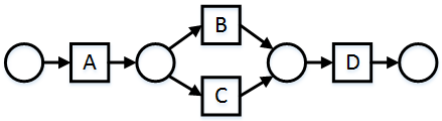

Fig. 1. An example of a Petri net

Table I shows an event log corresponding to a model from figure 1.

TABLE I. An example event log

Case	Events
0	A, B, D
1	A, B, C, D
2	A, C, D
3	A, B, D
4	A, C, D

Places in a Petri net may contain a number of tokens. Marking of a net is a distribution of tokens over the places. Marking represents a state of a Petri net. Any transition in a Petri net may fire if it is enabled, i.e. there are sufficient tokens in all of its input places. A firing of a transition is a single step in a modelled process, i.e. execution of an activity. When transition fires, it consumes tokens from input places, and produces token in its output places. Simultaneously with the transition firing an event record is added to a log. For a Petri net one can consider initial (starting) and final (ending) markings.

Process miners and developers of new algorithms for process discovering and analysis are interested in example-generation instrument. These people are core users for the

presented approach. Our tool has been developed as a plug-in for ProM 6 framework using Java 7 Standard Edition. The main features of the presented plug-in are:

- A user can easily generate a set of event logs with additional noise.
- Generation settings allow users to decide how many event logs will be generated, how many traces will these logs include. In order to prevent loops which will not terminate, user is asked about a maximum number of steps during algorithm execution. All event logs will be generated within one execution of the plug-in. By default the tool generates 5 event logs while every log consists of 10 traces and it does at most 100 steps.
- In cases when several outputs from one place are available it gives the possibility for flexible modifications of simulated behaviour which bring the higher accuracy of model behaviour describing the real world processes.
- It is possible to separate the start of a transition and the termination of a transition in event log records. Furthermore, in such cases users can define time of execution for every transition and how accurate they are executed by defining deviations bounds.
- The tool can create both event logs which completely fit the given model, and the logs with noise added.

We decided to implement our approach in the form of ProM 6 framework plug-in. The framework already has plug-ins which take care of visualization for Petri nets, event logs import and export, compatibility of logs with miner plug-ins, and provide further opportunities to work with resulting data. It was not necessary to develop additional supporting software.

B. Approach

This section describes an approach for log generation proposed in this work. Our approach contains three main parts: (1) simple log generation, (2) generation of a set of event logs, and (3) adding of an artificial noise. In the following we will consider all these parts.

1) Generation of an event log: This subsection describes simple log generation process. In order to generate a case in an event log the tool does the following steps:

(a) Adds tokens to all places from the initial marking.

(b) Creates an empty set which will be used to store the places with tokens. At this step only initial places have already obtained a token so they are added to the set.

(c) Next step is to select from which place we will try to fire a transition. It is handled by randomly picking a place from the set of places with tokens. Our algorithm does it without looking whether this place has outputs which could be fired or not. We do it in a way that prevents the looping in a situation when a place has a token, available outputs, but these outputs eventually lead to the same place without any other possible ways.

(d) The chosen place checks whether it has available outputs. If this is the case, an output will be chosen and fired according to priorities of available outputs. Looking for the place available is done in the following way: (1) we iterate through inputs and try to hold one token from every input; (2) if we meet an input from which we cannot hold a token, it means that this output is not available; (3) otherwise, an output is available; (4) in both cases we release held tokens.

(e) Firing of a transition implies the following steps: (1) information about the event is recorded into an event log (according to the chosen settings of noise generation and timing mode); (2) tokens are added to all places which are located as outputs for this transition; (3) a set of places which got tokens is returned.

(f) Then we check if any of final places got a token. In

```
Data: Initial marking as initialMarking,
Final marking as finalMarking,
Settings as settings.
Result: Event log
log = ∅;
time = getCurrentTime();
index = 1;
while index ≤ settings.numberOfTraces do
    trace = ∅;
    initialPlaces = initialMarking.getPlaces();
    finalPlaces = finalMarking.getPlaces();
    foreach place in initialPlace do
        place.addToken();
    end
    placesWithTokens = ∅;
    placesWithTokens.addAll(initialPlaces);
    step = 0;
    hasFinished = false;
    while step < settings.numberOfSteps AND NOT
    hasFinished do
        // Chooses place using random number
        currentPlace = choosePlace(placesWithTokens);
        // Tries to move from this place, if it is possible
        // moves, makes record about it in the trace
        // and returns set of places which got tokens
        newPlacesWithTokens =
        currentPlace.move(trace, time);
        removePlacesWithoutTokens(placesWithTokens);
        foreach place in finalPlaces do
            if place.getNumberOfTokens() > 0 then
                hasFinished = true;
                break;
            end
        end
        placesWithTokens.addAll(newPlacesWithTokens);

        step = step + 1;
    end
    foreach place in placesWithTokens do
        place.deleteAllTokens();
    end
    log.add(trace);
    index= index + 1;
end
return log;
```

Algorithm 1: An event log generation method

this case we finish the evaluation, otherwise we do the next two steps: (1) places from the original set which have no more tokens are eliminated, and (2) two sets of places are joined.

(g) Evaluation ends with deleting tokens from places which have them.

Algorithm 1 shows more precise and formal schema of the general log generation method.

```
Data: Initial marking as initialMarking, Final marking
      as finalMarking, Settings as settings.
Result: Event Log Array
i = 0;
eventLogArray = ∅;
while i < settings.numberOfLogs do
    log = generateLog(initialMarking, finalMarking,
    settings);
    eventLogArray.add(log);
    i = i + 1;
end
return eventLogArray;
```

Algorithm 2: Generation of a set of event logs

2) Generation of a set of logs: Multiple log generation is one of the main features of the tool presented. To generate a set of logs the tool is using a loop which is called generateLog(). Every time we use it we generate one log. So we repeat it until we get the desired number of logs. If the initial marking contains several places and a set of initial places to be randomly selected, each execution of log generation method works with it's own start. A set of event logs is stored in an object of EventLogArray class. This is the special class from ProM 6 Divide and Conquer package [12] intended to store the sets of event logs. As previously mentioned, ProM framework has a modular structure and contains lots of plug-ins for different operations [6]. Several plug-ins contain classes and methods which support the work with particular modelling formalisms and approaches. In our work we use one of these common classes to work with sets of event logs. Thus, one can process the generated sets directly in other plug-ins which are based on Divide and Conquer package. Algorithm 2 is intended to generate a set of event logs.

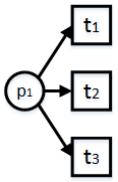

Fig. 2. Priorities

3) Priorities: In the case when a place has multiple output arcs the plug-in allows users to decide which output is more likely to be fired. Every output has a so-called priority which resembles the probability of this output to be selected. Each output can have a priority between 0 and 100 (including 0 and 100). Zero priority means that this output arc will be completely ignored. However, maximum priority does not mean that this output will be always fired. For every 2 outputs o_1 and o_2 it is true that relationship of the o_1 probability to be fired to the o_2 probability is equal to the relationship of o_1 priority to o_2 priority. Hence the higher priority, the higher chance for this output to be fired. Outputs with the same priority have equal chances to be fired. Algorithm 3 shows the approach.

Lets consider the example shown in figure 2. Consider the outputs (from p_1 to t_1), (from p_1 to t_2), (from p_1 to t_3) which have the priorities a, b, c respectively. Plug-in creates an array with a size of 3 elements. First element is equal to a, second is equal to $a+b$, the third is equal to $a+b+c$. Then plug-in gets a random number within a range from 0 to $a+b+c$ (excluding 0 and including $a+b+c$). If this random number is less or equal to a then the output (from p_1 to t_1) is fired. If the number is bigger than a, but less or equal to $a+b$ then the output (from p_2 to t_1) is fired, otherwise the fired output is (from p_1 to t_3).

```
Data: List of available outputs as availableOutputs.
Result: Output transition
// Creation of array whose length is
// equal to the length of availableOutputs
priorities;
if priorities.length > 0 then
    priorities[0] = availableTransitions[0].priority;
    i = 1;
    while i < priorities.length do
        priorities[i] = priorities[i - 1] +
        availableTransitions[i].priority;
        i = i + 1;
    end
    if priorities[priorities.length - 1] = 0 then
        return NULL;
    end
    randomNumber = getRandomNumber(0,
    priorities[priorities.length - 1]) + 1;
    i = 0;
    while i < priorities.length do
        if randomNumber ≤ priorities[i] then
            return availableOutputs[i];
        end
        i = i+1;
    end
end
```

Algorithm 3: Selection of a transition to fire

4) Noise adding: Noise is defined as deliberate deviations of generated event logs from a model real behaviour. If noise is applied a user is asked to select the so-called noise level. *Noise level* shows the probability of adding noise events to a log.

In the real-life processes noise usually consists of two components. First one is a totally chaotic represents interferences or crashes. Second one has more or less strict order. This component represents breakages, incorrect or unfair activities. In this work both components are taken into account.

Noise event can be represented in several ways:

- adding artificial transitions (with names specified by user);

- adding existent transitions from a model in incorrect order;
- skipped events; in such a case artificial events and existing transitions may be added to a log.

Thus, noise is added during the log generation process. Many tools try to add noise in a totally correct event log which already generated by some instrument or obtained from any system. Another way is to change the original model and to generate correct event logs from this changed model. Our scheme is more similar to real-life process execution. We generate logs with drawbacks and deviations during process functioning, which is usual for a number of processes. Table II shows an event log corresponding to a model from figure 1 with noise added.

TABLE II. AN EVENT LOG WITH NOISE

Case	Events
0	A, B, B, D
1	A, D, B, C, D
2	A, C, D
3	A, B, D
4	A, D

C. How to use the tool

The plug-in presented has several UI screens to interact with user. We provide a number of screenshots in order to illustrate our tool and make it easier for user to begin using it. The main screen asks a user about general settings of log generation (see figure 3). User may select *desired number of logs to be created*, *number of traces per each log*, *maximum number of algorithm steps for one trace*. In case when user uses noise generation, this is not a number of events per trace: some activities may be skipped, others may be added. User may specify to the plug-in *to use (or not) priorities* and/or *noise*. The screens specifying noise generation options are shown to the user, if he (or she) selects to use generation with noise.

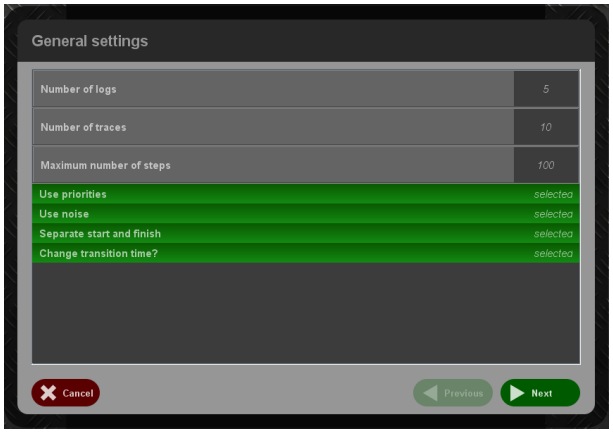

Fig. 3. First screen: General settings

User may specify if an execution of an activity is represented in a log with 1 or 2 events. Each transition is writen as 2 log events if it is important to separate whent execution of the transition starts and finishes. In such a case the first event indicates when the execution of the activity begin, whereas the second one indicates when it ends recording information about time according to specified time of execution.In addition, if user chooses to separate the start and complete events for each activity, it is possible to set the time of execution of every activity manually or skip it and use default values.

One of the screens demonstrates to a user the Petri net given as plug-in input (see figure 4). It uses visualization plug-in from the Petrinet package to show the model. User can use this screen to specify simulation settings more simply. This is favourable for Petri nets of any size.

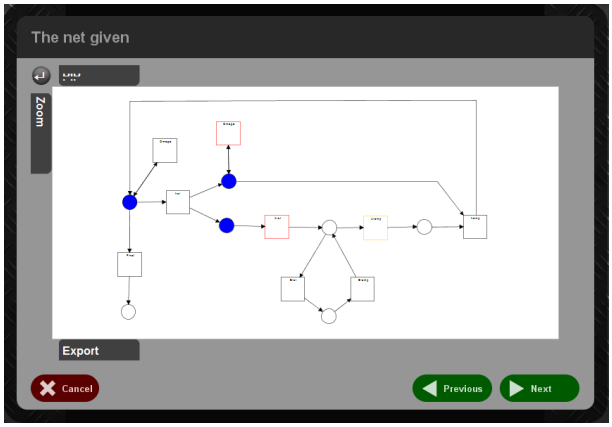

Fig. 4. Screen demonstrates a model given as plug-in input

Two screens ask user to pick an initial and final markings. User selects initial places from which an initial marking consists. The plug-in uses a final marking to end the simulation. Once a token is added to any of final places, the execution ends.

Some screens are optional. Series of screens help user to specify priorities for each place with undetermined output. Noise settings also may be specified with special screens. Users may specify the noise level and which kind of noise will be used. There are two possibilities: use only transitions of the given net, or add additional artificial transitions. Screen shown in figure 5 allows user to choose any number of transitions from a model given to appear in event log as noise transitions. Another screen helps to assign the set of artificial noise transitions. Screen with time settings allows to specify execution time for every transition (including artificial noise transitions) and maximum deviation from this time allowed for noise generator. We do not give all the screens here to not clutter up the text.

D. The tool and ProM 6 framework

This section presents an overview of the ProM 6 architecture. ProM is an open-source framework for implementation of the process mining algorithms in a standard environment. ProM consists of disjointed parts to increase the flexibility. The core part of the framework is distributed under GNU Public License. One may to upload plug-ins developed in the specific way and to work with them. The framework takes care about parameters needed for plug-ins. Special plug-ins were developed which load data into the environment and export results to disc as well as being stored in ProM resource pool for using them in other plug-ins. Almost all data types typical

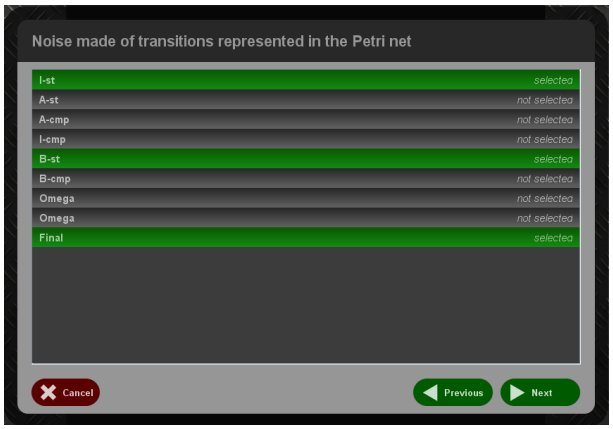

Fig. 5. Noise settings

for working with Petri nets have visualizers, so researchers and developers do not have to spend time on creating them.

Plug-ins may run with GUI or without it. It is allowed to call the other plug-ins or employ data types, visualization, import and export methods from a plug-in via special plug-in manager. The framework manages this usage in such a way: our plug-in sends a request to execute code from another package. The plug-in manager processes this request and returns an instance of the plug-in called. For example, model on screen shown in figure 4 visualized by a standard visualizer from the Petri nets package. In the future work we plan to improve this screen by using self-engineered visualizer which allows to set up several generation options (like priorities) directly on a model visualized.

Plug-in manager enables not only to call the known plug-ins but also to look for the plug-ins with specific signature, to call it, and to get results of its execution. Work with plug-ins based on such named contexts. Each plug-in must have a context. It may use the data objects within the context. For every context one can make a child context. Thus, it is possible to construct hierarchy of plug-in calls from a one parent plug-in.

The framework allows users to take an advantage of reusing previous executions of plug-ins via mechanism of so-called connections. In fact, connection is an object which holds a number of data objects in weak hash map. Connection can be reached after its registration in a framework context by any other plug-ins using connection manager. Connection manager takes one argument and searches for all the connections which hold specific parameter. The mechanism of connections allows to process data obtained by one plug-in by another one.

The core part of a typical ProM processing plug-in is a class which contains at least one public method. This class needs to contain at least one method with special annotation which registers it in the ProM framework as a plug-in. The name, input and output parameter lists are also listed inside the annotation. Particular plug-in context of a current ProM session should be among the other parameters.

The tool which implements an approach presented in this work is built as a plug-in for the ProM Framework, therefore architecture of the tool had to fulfil all requirements of ProM plug-ins listed above. Our tool consists of 6 main classes:

- *LogGenerator* class is responsible for interaction with framework and GUI.
- *AbstractPetriNode* represents an element of Petri net (place or transition). It wraps an object of PetriNode class from PetriNets package providing convenient access to inputs and outputs of a node.
- *Place* extends *AbstractPetriNode* getting specific features of a place. An object of this class holds a number of tokens and allows to choose between the outputs.
- *Transition* also extends *AbstractPetriNode* getting specific features of a transition. Actions of transition firing and writing to an event log are described in this class.
- *Generator* class encapsulates creation of a net acceptable for the log generation based on a given Petri net and performs generation.
- Object of *GenerationDescription* class holds information about settings specified by a user about the set of event logs to be generated (number of event logs, number of traces per log, priorities and others).

E. Example of the tool usage

Figure 6 shows several examples generated by our approach. In the first line (*a)*, *b)*, *c)*) original models are shown.

To examine our approach for each model were generated sets of event logs with different noise levels and generation settings. In figure 6 shown the model discovered by process discovery algorithm [1] from only one model for each set generated using 5% and 20% noise levels. Second line shows the models obtained using alpha algorithm [1] from the event log generated using 5% noise level. Level of 20% was used for the event logs from which the models in the third line were obtained.

In the first case (see *d)*, *g)*) transitions $A - st$, $B - st$ and $Final$ were used as noise transitions. Transitions $e1$, $e3$ and $e6$ are used as noise transitions in the second case (see *e)*, *h)*). In the third case (see *f)*, *i)*) transitions c and e were used as noise transitions. Artificial noise transitions were used in all cases: $noise1$, $noise2$, $noise3$. Transition skipping was enabled.

We do not show the models obtained from the logs with 0% noise level. Such a model is totally identical to the original one if sufficient number of traces is used. Process discovery algorithms may show strange or inappropriate results for a tiny event logs generated from the complex models.

Resulting model complexity highly depends on structure of original model used for generation and noise setting. For example, one can choose to add lots of artificial transitions to a log. Such setting leads to a generation of an event log from which one can obtain very sophisticated model.

In our example models obtained from event logs with 5% noise level differ from original models only in several actions. Whereas in cases with 20% models are more complex and different from original ones. It is useless to generate logs using total noise levels of 50% or more: one obtains chaotic

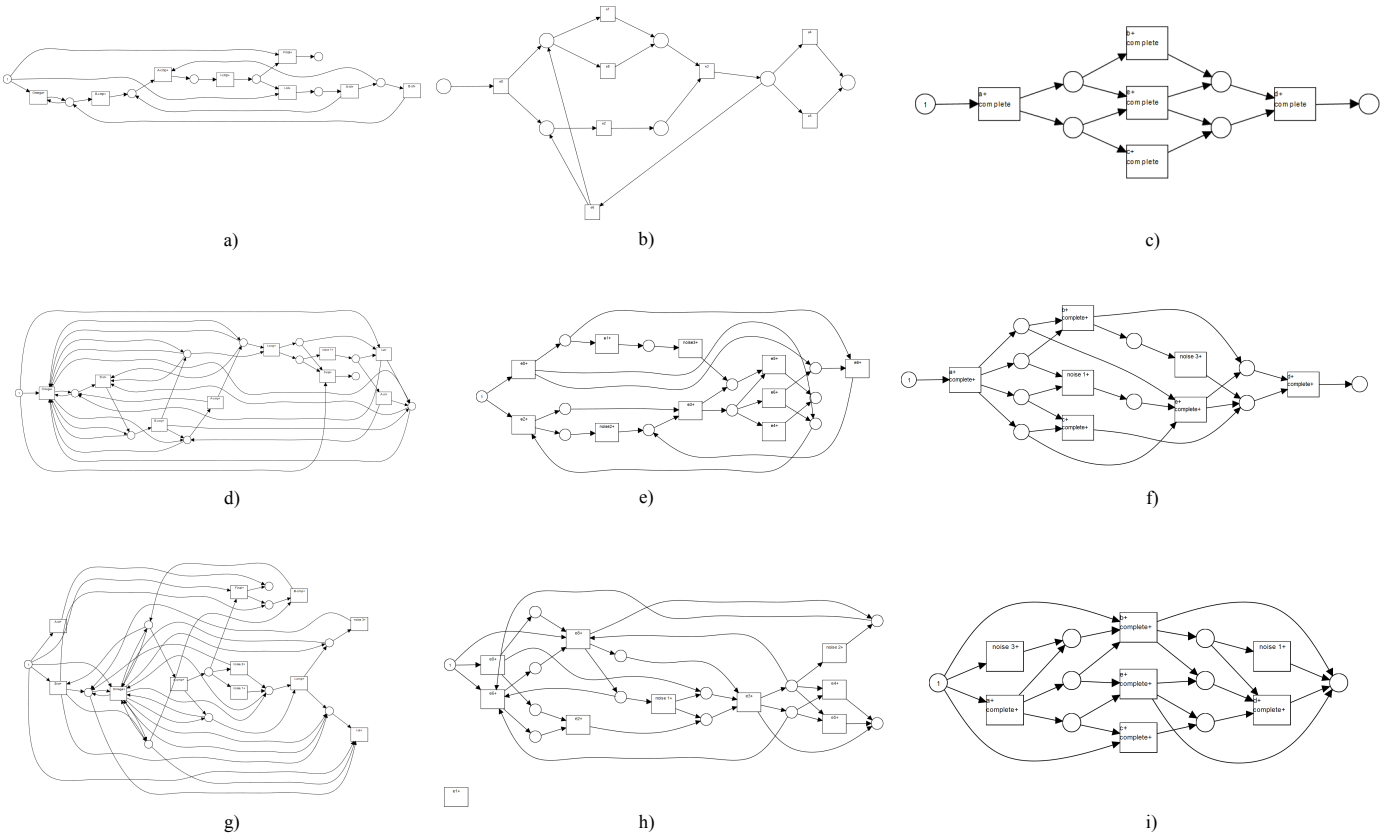

Fig. 6. Examples of models discovered from the event logs generated by the approach presented

behaviour totally different from the behaviour of an original model.

IV. Conclusion

In this paper we have presented an approach for generation of sets of event logs. This approach is implemented as a ProM 6 framework plug-in which can be easily used by process miners, researchers, and developers. It allows not only to generate the simple event logs, but also to generate a set of event logs, or event logs with noise. All these functions allow to run experiments in the relatively easy way with different algorithms implemented as a ProM plug-ins. Generated logs can be exported using standard ProM plug-ins to use them in other applications. Noise generation is also quite useful during plug-in testing process.

The tool presented takes into account the advantages and drawbacks of other existing approaches. Nevertheless, it also has its areas to improve. In the future work authors plan to deal with a generation of logs with additional resources. Another future development is the incorporation of different model formalisms into existing plug-in in addition to the Petri nets. Several improvement should be done in graphical user interface to simplify interaction with plug-in.

Acknowledgment

This work is output of a research project implemented as part of the Basic Research Program at the National Research University Higher School of Economics (HSE). Authors would like to thank all the colleagues from the PAIS Lab whose advice was very helpful in the preparation of this work.

References

[1] Wil M. P. van der Aalst, *Process mining: discovery, conformance and enhancement of business processes*. Springer, 2011.

[2] IEEE Task Force on Process Mining, "Process mining manifesto," in *Business Process Management Workshops*, ser. Lecture Notes in Business Information Processing, F. Daniel, K. Barkaoui, and S. Dustdar, Eds., vol. 99. Springer-Verlag, Berlin, 2012, pp. 169–194.

[3] W. M. P. v. d. Aalst, A. J. M. M. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 9, pp. 1128–1142, 2004.

[4] A. Rozinat and W. M. P. v. d. Aalst, "Conformance testing: Measuring the fit and appropriateness of event logs and process models," in *Business Process Management Workshops*. Springer, 2006, pp. 163–176.

[5] B. F. van Dongen, W. M. P. van der Aalst, C. W. Günther, A. Rozinat, E. Verbeek, and T. Weijters, "ProM: the process mining toolkit," in *Business Process Management Demonstration Track (BPMDemos 2009)*, ser. CEUR Workshop Proceedings, A. K. A. d. Medeiros and B. Weber, Eds., vol. 489. Ulm, Germany: CEUR-WS.org, 2009, pp. 1–4.

[6] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, "Prom 6: The process mining toolkit," *Proc. of BPM Demonstration Track*, vol. 615, pp. 34–39, 2010.

[7] W. M. P. van der Aalst, "Decomposing petri nets for process mining: A generic approach," *Distributed and Parallel Databases*, vol. 31, no. 4, pp. 471–507, 2013.

[8] W. M. P. van der Aalst, "Mine your own business: Using process mining to turn big data into real value," in *Proceedings of the 21st European Conference on Information Systems (ECIS 2013)*. Utrecht, The Netherlands: AIS Electronic Library, 2013, pp. 1–9.

[9] A. K. A. d. Medeiros and C. W. Günther, "Process mining: Using CPN tools to create test logs for mining algorithms," in *Proceedings of the Sixth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2005)*, ser. DAIMI, K. Jensen, Ed., vol. 576. Aarhus, Denmark: University of Aarhus, 2005, pp. 177–190.

[10] A. Burattin and A. Sperduti, "PLG: a framework for the generation of business process models and their execution logs," in *BPM 2010 Workshops, Proceedings of the Sixth Workshop on Business Process Intelligence (BPI2010)*, ser. Lecture Notes in Business Information Processing, J. Su and M. z. Muehlen, Eds., vol. 66. Springer-Verlag, Berlin, 2011.

[11] T. Stocker and R. Accorsi, "Secsy: Security-aware synthesis of process event logs," in *Proceedings of the 5th International Workshop on Enterprise Modelling and Information Systems Architectures*, St. Gallen, Switzerland, 2013.

[12] E. Verbeek and W. M. P. v. d. Aalst, "Decomposing replay problems: A case study," in *Joint Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'13) and the International Workshop on Modeling and Business Environments (ModBE'13), Milano, Italy, June 24 - 25, 2013*, ser. CEUR Workshop Proceedings, D. Moldt, Ed., vol. 989. CEUR-WS.org, 2013, pp. 219–235. [Online]. Available: http://ceur-ws.org/Vol-989/paper07.pdf