Simulating Petri Nets with Inhibitor and Reset Arcs

P.A. Persukhov, ORCID: 0000-0003-1923-976X <papertsukhov@edu.hse.ru>
A.A. Mitsyuk, ORCID: 0000-0003-2352-3384 <amitsyuk@hse.ru>
PAIS Lab, Faculty of Computer Science, National Research University Higher School of Economics,
3 Kochnovskiy Proezd, Moscow, 101000, Russia.

Abstract. Event logs of software systems are used to analyze their behaviour and inter-component interaction. Artificial event logs of software systems are used to analyze their behaviour and inter-component interaction. Recent methods allow to generate artificial event logs by simulating ordinary Petri nets. In this paper we present the algorithm generating event logs for Petri nets with inhibitor and reset arcs. Nets with inhibitor arcs are more expressive than ordinary Petri nets, and allow to conveniently model conditions in real-life software. Resets are common in real-life systems as well. This paper describes the net simulation algorithm, and shows how it can be applied for event log generation.

Keywords: Petri nets; inhibitor arcs; reset arcs; simulation; event logs

For citation: Persukhov P.A., Mitsyuk A.A. Simulating Petri Nets with Inhibitor and Reset Arcs. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 151-162. DOI: 10.15514/ISPRAS-2019-31(4)-10

Acknowledgments. This work is supported by the Basic Research Program at the National Research University Higher School of Economics.

Simulating Petri Nets with Inhibitor and Reset Arcs

Simulating Petri Nets with Inhibitor and Reset Arcs.

P.A. Перцухов, ORCID: 0000-0003-1923-976X <papertsukhov@edu.hse.ru>
A.A. Мицюк, ORCID: 0000-0003-2352-3384 <amitsyuk@hse.ru>
Лаборатория ПОИС, факультет компьютерных наук, Национальный исследовательский университет «Высшая школа экономики», 101000, Россия, Москва, Кочновский проезд, 3

Abstract. Журналы событий программных систем используются для анализа их поведения и взаимодействия между компонентами. Искусственные журналы событий с подходящими свойствами необходимы для тестирования алгоритмов, используемых для такого анализа. Современные методы позволяют генерировать искусственные журналы событий в результате симуляции обычных сетей Петри. В этой статье мы рассматриваем алгоритм, генерирующий журналы событий для сетей Петри с ингибиторными дугами и дугами сброса. Эти дуги позволяют генерировать искусственные журналы событий, которые можно использовать для тестирования алгоритмов, используемых для такого анализа. Современные методы позволяют генерировать искусственные журналы событий в результате симуляции обычных сетей Петри. В этой статье мы рассматриваем алгоритм, генерирующий журналы событий для сетей Петри с ингибиторными дугами и дугами сброса. Сети с ингибиторными дугами более выразительны, по сравнению с классическими сетями Петри, и позволяют удобно моделировать условия в реальном программном обеспечении. Операции сброса также распространены в реальных системах. В этой статье описывается алгоритм симуляции сетей Петри с ингибиторными дугами и сбросами. Сети с ингибиторными дугами и сбросами позволяют генерировать искусственные журналы событий, которые можно использовать для тестирования алгоритмов, используемых для такого анализа. Современные методы позволяют генерировать искусственные журналы событий, которые можно использовать для тестирования алгоритмов, используемых для такого анализа. Сети с ингибиторными дугами и сбросами позволяют генерировать искусственные журналы событий, которые можно использовать для тестирования алгоритмов, используемых для такого анализа. Сети с ингибиторными дугами и сбросами позволяют генерировать искусственные журналы событий, которые можно использовать для тестирования алгоритмов, используемых для такого анализа. Сети с ингибиторными дугами и сбросами позволяют генерировать искусственные журналы событий, которые можно использовать для тестирования алгоритмов, используемых для такого анализа. Сети с ингибиторными дугами и сбросами позволяют генерировать искусственные журналы событий, которые можно использовать для тестирования алгоритмов, используемых для такого анализа.
2.2 Petri nets with Inhibitor and Reset Arcs

Petri nets also contain places, transitions, and ordinary control flow arcs. In this paper, we consider Petri nets with arcs of two additional types: reset and inhibitor arcs. These arcs are used to model situations where a transition can be inhibited or reset by other places.

Labels are placed inside the transition boxes. Petri nets can contain invisible (silent) process actions which are labelled with \( \tau \). Labels are assigned to transitions via a labelling function \( \lambda: T \rightarrow \mathcal{A} \cup \{\tau\} \).

A state of an ordinary Petri net is called its marking. It is a function \( M: P \rightarrow \mathbb{N} \) assigning natural numbers to places. In figures, a marking \( M \) can be designated by putting \( M(p) \) black tokens into a place \( p \) of the net. By \( M_0 \) we denote the initial marking. For example, the initial marking of the net from Fig. 1 consists of a single token in the place \( p_0 \).

A transition represents an activity of a process. It is enabled in a current marking if in each of its input places (for \( t \in T \) input places arc \( \bullet t = \{p \mid (p, t) \in F\} \)) there enough tokens, that is \( \forall p \in \bullet t: M(p) \geq 1 \). An enable transition may fire that changes a marking of the net. It consumes tokens from the input places, and produces tokens to output places (for \( t \in T \) input places arc \( t \bullet = \{p \mid (t, p) \in F\} \)).

Consider a model from Fig. 1: \( t_1 \) is the only enabled transition in the initial marking. It may fire, that corresponds to an occurrence of activity “A”. Then, the transition consumes a single token from \( p_0 \) and produces tokens to \( p_1 \) and \( p_2 \). Fig. 2 illustrates this firing. The firing is local, each transition fires independently from other transitions.

### 2.2 Petri nets with Inhibitor and Reset Arcs

In this paper, we consider Petri nets with arcs of two additional types: reset and inhibitor arcs. These arcs also contain places, transitions, and ordinary control flow arcs.

Places are shown by circles, transitions are shown by boxes, the flow relation is depicted using ordinary directed arcs. In fig. 1, there are three places \( (p_0, p_1, p_2) \) and four transitions \( (t_1, t_2, t_3, t_4) \). Transitions are labelled with activity names from the set \( \mathcal{A} = \{A, B, C, D\} \).

A labelled Petri net with weights, inhibitor and reset arcs (WIR Petri net) is a tuple \( N_{WIR} = (P, T, F, W, R, I, \lambda) \), where:

- \( (P, T, F) \) is an ordinary Petri net,
- \( W \subseteq F \rightarrow \mathbb{N} \) is an arc weight function,
- \( R \subseteq P \times T \) is a function defining reset arcs,
- \( I \subseteq P \times T \) is an inhibiting relation,
- and \( \lambda: T \rightarrow A \cup \{\tau\} \) is a labelling function.

Fig. 3 shows an example WIR Petri net.

A reset arc removes all tokens from the place no matter of their number. These arcs also called clear arcs [14].

In Fig. 3, there is a reset arc from the place \( p_1 \) to the transition \( t_3 \) labelled with “C”. Reset arcs are denoted with double arrows at the end. Note that the net contains a loop of two actions “A” (\( t_1 \)) and “D” (\( t_4 \)). The possible sequence of firings is \( t_1, t_4, t_3, t_4, t_1, t_3 \). Before the last step, \( p_1 \) will contain 3 tokens all of which will be removed by the firing of \( t_3 \).

An inhibitor arc [15], [16] can be from a place to a transition. This transition cannot fire if there is a token in place connected with the transition using an inhibitor arc. For an inhibitor arc \( o t = \{p \mid (p, t) \in I\} \) denotes the set of inhibiting places for \( t \). That is, an inhibiting place allows to prevent the transition firing. Transitions consume no tokens through inhibitor arcs.

Inhibitor arcs are shown with small circles instead of arrows at the end. In Fig. 3, there is a reset arc from the place \( p_4 \) to the transition \( t_2 \) labelled with “B”. Thus, \( t_2 \) can not fire if there is a token in \( p_4 \) no matter how many tokens are in \( p_2 \). The whole part of the model that consists of places \( p_3 \) and \( p_4 \), transitions \( t_2 \) and \( t_6 \) is a switch with two possible states: open (there is a token in \( p_3 \)) and closed (there is a token in \( p_4 \)).

The firing of \( t_6 \) closes the switch and deprecates the firing of \( t_2 \). Thus, only \( t_3 \) is able to clear tokens from the place \( p_1 \) if there is a token in \( p_2 \). The firing of \( t_3 \) will end the process, because it will consume and remove all tokens from the upper part of the net.

The firing of \( t_6 \) opens the switch. Then, \( t_2 \) may consume tokens from \( p_1 \) independently of tokens in other places. Note that the arc from \( p_1 \) to \( t_2 \) has a weight of 2. Thus, each firing of \( t_2 \) consume exactly 2 tokens, and \( t_2 \) can not fire if there is only one token in \( p_1 \). Note that a particular WIR Petri net can contain zero number of reset and inhibitor arcs.

Marking of a WIR Petri net is defined in the same way as for an ordinary Petri net. But the firing rule is slightly different for WIR Petri nets. Each marking change is called step. In this paper, we
assume that each step consists of a single transition firing. The step from Fig. 2 is denoted by
\( M_0(t_1)M' \), where \( M'(p_1) = M'(p_2) = M'(p_3) = 1 \) and \( M'(p_4) = 0 \).

2.3 Event Logs

In this paper, we apply process model simulation to generate event logs with records of the behavior. We define an event log as a finite multiset of traces \( \mathcal{B}(\mathcal{A}) \). A trace \( \sigma \in \mathcal{A}^* \) is a finite sequence of events from the set \( \mathcal{A} \). Note that transitions of a WIR Petri net are labelled with elements of \( \mathcal{A} \). The only transitions which firing leaves no events are silent \( r \)-transitions.

Technically, we record the event logs in XES (Extensible Event Stream) format\(^2\) that will be considered in more detail in Section 4.

3. Petri Net Simulation Algorithm

This section describes the algorithm to simulate labelled WIR Petri nets. The main idea of the algorithm is to iterate over all transitions and fire one of them at each iteration, recording corresponding events to the log. This procedure is performed in the main generating function called \( \text{generateTrace} \) (see Algorithm 1).

Input: transitions, initialMarking, finalMarking, settings as \( \{ \text{maxNumberOfSteps}, \text{maxIterations}, \text{isRemovingUnfinishedTraces} \} \)

Output: generated trace or NULL

1: function \( \text{generateTrace} \)
2: trace <- NULL;
3: replayCompleted <- false;
4: addTraceToLog <- false;
5: stepNumber <- 0;
6: repeat
7:     moveToInitialState();
8:     trace <- createTrace();
9:     stepNumber <- 0;
10:     while stepNumber < maxNumberOfSteps
11:         and not replayCompleted do
12:             transition <- chooseNextTransition();
13:             if transition = NULL then
14:                 break;
15:             end if
16:             fire(transition, trace);
17:             replayCompleted <- isCompleted(finalMarking);
18:             stepNumber <- stepNumber + 1;
19:         end while
20:     iteration <- iteration + 1;
21:     until (iteration >= maxIterations or not isRemovingUnfinishedTraces or replayCompleted)
22:     if not replayCompleted
23:         and isRemovingUnfinishedTraces then
24:             trace <- NULL;
25:         end if
26: end function

Algorithm 1. One trace generation

This algorithm works as follows. We have \( \text{maxIterations} \) attempts to reach the final marking of the net. By default this number is 10. The function \( \text{moveToInitialState} \) initiates the trace generation by setting a marking of the to \( M_0 \). Then, we create an empty trace by calling \( \text{createTrace} \).

At each step of the main loop, the algorithm chooses an enabled transition in the \( \text{chooseNextTransition} \), and fire it (function \( \text{fire} \)). The function \( \text{fire} \) changes a marking of the net and writes an event to the trace. Then we call the function \( \text{isCompleted} \) to check if we reached the final marking and update \( \text{replayCompleted} \). This loop iterates until we reach the final marking (\( \text{replayCompleted} = \text{true} \)) or exceed the specified limit of steps for one trace \( \text{maxNumberOfSteps} \).

When we cannot find an enabled transition which is ready to fire, we clear the unfinished trace and begin a new attempt.

If no one of 10 attempts succeeded, we return NULL which will be recorded as an empty trace to the event log. Note that there is a setting of the prototype tool that removes all empty and unfinished traces.

The \( \text{chooseNextTransition} \) function selects an enabled transition using a specified rule. The most basic implementation of this function is shown in Algorithm 2. Here, the random transition among all enabled and noise transitions is selected.

Input: allTransitions, noiseTransitions

Output: selected transition or NULL

1: function \( \text{chooseNextTransition} \)(allTransitions, noiseTransitions)
2: enabledTransitions <- findEnabledTransitions();
3: return random transition among enabledTransitions or NULL;
4: end function

Algorithm 2. Looking for the next transition

This algorithm is based on the algorithm for ordinary Petri nets [17], and thus, is able to add noise to the event log. More complex rules to select the enabled transition can be applied. For example, priorities of preferences can be assigned to the transitions which affect the order of their firing. If there is no enabled transition, then NULL is returned.

To check if a transition \( t \) is enabled, we ensure that all input places connected with \( t \) with the help of ordinary arcs have enough tokens. Besides that, we check that places connected with \( t \) with the help of inhibitor arcs don’t contain any tokens. Reser arcs don’t affect if a transition is enabled or not. Algorithm 3 shows how this is done.

Input: allTransitions

Output: list of enabled transitions

1: function \( \text{findEnabledTransitions} \)(allTransitions)
2: enabledTransitions <- ∅;
3: for transition in allTransitions do
4:     enabled <- true;
5:     for arc in transition.inputArcs do
6:         if arc.place.numberOfTokens < arc.weight then
7:             enabled <- false;
8:         end if
9:     end for
10: end function

Algorithm 3. Finding the enabled transitions
Pertsukhov P.A., Mitsyuk A.A. Simulating Petri Nets with Inhibitor and Reset Arcs. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019, pp. 151-162.

```java
12: if not enabled then
13:     continue;
14: end if
15: for arc in transition.inhibitorArrows do
16:     if arc.place.numberOfTokens > 0 then
17:         enabled ← false;
18:         break;
19: end if
20: end for
21: if enabled then
22:     enabledTransitions.add(transition);
23: end if
24: end for
25: return enabledTransitions;
26: end function
```

**Algorithm 3. Finding enabled transitions**

Algorithm 4 shows the transition firing function. This function produces and consumes tokens, and then adds an event corresponding to this transition to the trace. The basic implementation is shown which considers only transition names. There are much more complicated implementations of this function for time-driven, resources, and priorities generation modes.

```java
1: function fire(transition, trace)
2:     for arc in transition.inputArrows do
3:         arc.place.consumeToken(arc.weight);
4:     end for
5:     for arc in transition.inputResetArrows do
6:         arc.place.consumeAllTokens();
7:     end for
8:     log transition to trace or perform some noise event
9:     for arc in transition.outputArrows do
10:        arc.place.produceTokens(arc.weight);
11:    end for
12: end function
```

**Algorithm 4. Firing function**

### 4. Prototype Tool

The presented event log generation algorithm has been implemented as a prototype tool. It is written in Java and Kotlin programming languages. In this section, we consider the tool. The tool consists of two parts: Generation Setup unit and Generation unit.

#### 4.1 Preparing for generation

In preparation part we receive settings from the GUI (see fig. 4) or read a JSON (see fig. 5) file. Settings from JSON are validated. Then we load the model from a PNML file and prepare this model for generation. Inhibitor and reset arcs could be either specified in settings file, or loaded from the PNML file. The initial and final markings are loaded in the same manner.

---

<Fig. 4. Tool GUI>

<Fig. 5. Generation settings in JSON>
After that we create an instance of special class `GenerationHelper`, which encapsulates the main code for choosing transitions, looking for enabled transitions, handling noise and artificial log events if it is needed. There are different helpers for simple generation, generation with priorities, and generation with time.

Also, we convert each transition to a special loggable transition-related class which is used during generation. They contain methods of event recording to a trace. Such a class also consumes tokens from input places and produces tokens to output places. Methods to check if a transition is enabled are also here.

4.2 Preparing for generation

A singleton class is used to record the event log. We setup this logging class. It uses the OpenXES\(^4\) library with the help of which we can write XES log files. The XES format is common for the field of process mining [1]. OpenXES library creates a separate file for each log. A fragment of the example output in XES file is shown in fig. 6. This example contains two traces with names “Trace 4” and “Trace 5”. These names should be unique.

The first trace is of the three events: “B”, “D”, and “D”. Note that XES is XML-based and is easy-to-read for a machine or a human. Then we use a `Generator` class to launch the main generation method (see Algorithm 1), passing a `generationHelper` to this class.

![Fig. 6. Fragment of XES file](image)

4.3 Tool Usage

Let us test our tool on the model from the fig. 7.

The initial marking consists of four tokens (shown as black dots): one token lies in `place2`, one is in `place5`, and two tokens are in `place1`. Our goal is to reach the final marking (shown as green dots): two tokens in `place7`, and one token in `place6`.

![Fig. 7. Petri net used for log generation](image)

Transition C is enabled only when place place6 is empty. C consumes 2 tokens from place4 and produces a token to place7. Transition B removes all tokens in places place1 and place2. When this transition fires, it consumes 4 tokens from place4 and produces a token to place7. Thus, B can not fire before A. At each step either X or Y is enabled, so these transitions may produce a trace of any length. We set-up our tool to remove unfinished traces and limited max number of steps to 10. The result of the generation is shown in fig. 8.

![Fig. 8. Resulting traces](image)

Let us look at some trace, for example: `<A, X, Y, B, Y, C, X>`. First fired transition was A, and it produced 6 tokens to place4. Then X fired. It consumed a token from place5 and produced one to place6. C cannot fire in this marking. Thus, B fires. 4 of 6 tokens was consumed from place4, and one token produced to place7. Then Y fired and «opened» C. C has been fired just after Y was fired.

---

\(^4\) http://www.xes-standard.org/openxes/start
when a token has been removed from place6. B and C were fired once each, and produced 2 tokens in total to place7. place1 and place2 were cleared by B. The last event was X, which placed a token to place6. At the end of this run all tokens reside in the final marking.

5. Conclusion
In this paper, we have presented the algorithm to simulate a process model in the form of weighted labelled Petri net with inhibitor and reset arcs. This algorithm can be applied to generate event logs from the event log. Proposed algorithm continues the previous works on Petri net simulation with the purpose of generating artificial event logs. The prototype implementation is based on Gena tool\textsuperscript{3}. We have plans for future work. Firstly, we plan to comprehensively evaluate the proposed algorithm on artificial and real-life process models. For now, we just tested it on sample models to check algorithm validity. Secondly, we also plan to improve the prototype implementation and make it stable and usable. Thirdly, Gena is able to simulate timed process models, models with resources, data, add noise to an event log\textsuperscript{8},\textsuperscript{17}. Recently, an extension for Gena to simulate the multi-agent system has been proposed\textsuperscript{18}. We plan to merge these extensions and the algorithm presented in this paper. Then, it will be possible to simulate WIR Petri nets with time/resources, and data.

References
\textsuperscript{1} Wil M.P. van der Aalst. Process mining: Discovery, Conformance and Enhancement of Business Processes. Springer, 2011, 352 p.
\textsuperscript{2} J. Carmona, B.F. van Dongen, A. Solti, and M. Weidlich. Conformance Checking – Relating Processes and Models. Springer, 2018, 270 p.
\textsuperscript{3} M. Weske. Business Process Management: Concepts, Languages, Architectures. Springer, 2007, 408 p.
\textsuperscript{4} Wil M.P. van der Aalst. Process mining and simulation: a match made in heaven! In Proc. of the 50th Computer Simulation Conference, 2018, Article No. 4.
\textsuperscript{5} A. Burattin and A. Sperduti. PLG: A framework for the generation of business process models and their execution logs. Lecture Notes in Business Information Processing, vol. 66, 2010, pp. 214–219.
\textsuperscript{6} A. Burattin. PLG2: multiperspective process randomization with online and offline simulations. In Proc. of the BPM Demo Track 2016, CEUR Workshop Proceedings, vol. 1789, 2016.
\textsuperscript{7} T. Jouck and B. Depaire. PetAndLogGenerator: A generator for artificial event data. In Proc. of the BPM Demo Track 2016, CEUR Workshop Proceedings, vol. 1789, 2016.
\textsuperscript{8} A.A. Mitsyuk, I.S. Shugurov, A.A. Kalenkova, and W.M.P. van der Aalst. Generating event logs for high-level process models. Simulation Modelling Practice and Theory, vol. 74, 2017, pp. 1–16.
\textsuperscript{9} W. Resig. Understanding Petri Nets – Modeling Techniques, Analysis Methods, Case Studies. Springer, 2013, 260 p.
\textsuperscript{10} K. Jensen and L. M. Kristensen. Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer, 2009, 384 p.
\textsuperscript{11} I.A. Lomazova. Nested Petri Nets: Multi-level and Recursive Systems. Fundamenta Informaticae, vol. 47, no. 3-4, 2001, pp. 283–293.
\textsuperscript{12} R. Valk. Object Petri nets: Using the nets-within-nets paradigm. Lecture Notes in Computer Science, vol. 3098, 2003, pp. 819–848.
\textsuperscript{13} W.M.P. van der Aalst and K.M. van Hee. Workflow Management: Models, Methods, and Systems. MIT Press, 2002, 384 p.
\textsuperscript{14} C. Lakos and S. Christiansen. A general systematic approach to arc extensions for coloured Petri nets. Lecture Notes in Computer Science, vol. 815, 1994, pp. 338–357.
\textsuperscript{15} R. Janicki and M. Koutny. Semantics of inhibitor nets. Information and Computation, vol. 123, no. 1, 1995, pp. 1–16.
\textsuperscript{16} H. C. M. Kleijn and M. Koutny. Process semantics of pt-nets with inhibitor arcs. Lecture Notes in Computer Science, vol. 1825, 2000, pp. 261–281.
\textsuperscript{17} S. Shugurov and A.A. Mitsyuk. Generation of a Set of Event Logs with Noise. In Proc. of the 5th Spring/Summer Young Researchers Colloquium on Software Engineering (SYRCoSE 2014), 2014, pp. 88–95.
\textsuperscript{5} https://pais.hse.ru/research/projects/gena

5. Conclusion
In this paper, we have presented the algorithm to simulate a process model in the form of weighted labelled Petri net with inhibitor and reset arcs. This algorithm can be applied to generate event logs from the event log. Proposed algorithm continues the previous works on Petri net simulation with the purpose of generating artificial event logs. The prototype implementation is based on Gena tool\textsuperscript{3}. We have plans for future work. Firstly, we plan to comprehensively evaluate the proposed algorithm on artificial and real-life process models. For now, we just tested it on sample models to check algorithm validity. Secondly, we also plan to improve the prototype implementation and make it stable and usable. Thirdly, Gena is able to simulate timed process models, models with resources, data, add noise to an event log\textsuperscript{8},\textsuperscript{17}. Recently, an extension for Gena to simulate the multi-agent system has been proposed\textsuperscript{18}. We plan to merge these extensions and the algorithm presented in this paper. Then, it will be possible to simulate WIR Petri nets with time/resources, and data.

References
\textsuperscript{1} Wil M.P. van der Aalst. Process mining: Discovery, Conformance and Enhancement of Business Processes. Springer, 2011, 352 p.
\textsuperscript{2} J. Carmona, B.F. van Dongen, A. Solti, and M. Weidlich. Conformance Checking – Relating Processes and Models. Springer, 2018, 270 p.
\textsuperscript{3} M. Weske. Business Process Management: Concepts, Languages, Architectures. Springer, 2007, 408 p.
\textsuperscript{4} Wil M.P. van der Aalst. Process mining and simulation: a match made in heaven! In Proc. of the 50th Computer Simulation Conference, 2018, Article No. 4.
\textsuperscript{5} A. Burattin and A. Sperduti. PLG: A framework for the generation of business process models and their execution logs. Lecture Notes in Business Information Processing, vol. 66, 2010, pp. 214–219.
\textsuperscript{6} A. Burattin. PLG2: multiperspective process randomization with online and offline simulations. In Proc. of the BPM Demo Track 2016, CEUR Workshop Proceedings, vol. 1789, 2016.
\textsuperscript{7} T. Jouck and B. Depaire. PetAndLogGenerator: A generator for artificial event data. In Proc. of the BPM Demo Track 2016, CEUR Workshop Proceedings, vol. 1789, 2016.
\textsuperscript{8} A.A. Mitsyuk, I.S. Shugurov, A.A. Kalenkova, and W.M.P. van der Aalst. Generating event logs for high-level process models. Simulation Modelling Practice and Theory, vol. 74, 2017, pp. 1–16.
\textsuperscript{9} W. Resig. Understanding Petri Nets – Modeling Techniques, Analysis Methods, Case Studies. Springer, 2013, 260 p.
\textsuperscript{10} K. Jensen and L. M. Kristensen. Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer, 2009, 384 p.
\textsuperscript{11} I.A. Lomazova. Nested Petri Nets: Multi-level and Recursive Systems. Fundamenta Informaticae, vol. 47, no. 3-4, 2001, pp. 283–293.
\textsuperscript{12} R. Valk. Object Petri nets: Using the nets-within-nets paradigm. Lecture Notes in Computer Science, vol. 3098, 2003, pp. 819–848.
\textsuperscript{13} W.M.P. van der Aalst and K.M. van Hee. Workflow Management: Models, Methods, and Systems. MIT Press, 2002, 384 p.
\textsuperscript{14} C. Lakos and S. Christiansen. A general systematic approach to arc extensions for coloured Petri nets. Lecture Notes in Computer Science, vol. 815, 1994, pp. 338–357.
\textsuperscript{15} R. Janicki and M. Koutny. Semantics of inhibitor nets. Information and Computation, vol. 123, no. 1, 1995, pp. 1–16.
\textsuperscript{16} H. C. M. Kleijn and M. Koutny. Process semantics of pt-nets with inhibitor arcs. Lecture Notes in Computer Science, vol. 1825, 2000, pp. 261–281.
\textsuperscript{17} S. Shugurov and A.A. Mitsyuk. Generation of a Set of Event Logs with Noise. In Proc. of the 5th Spring/Summer Young Researchers Colloquium on Software Engineering (SYRCoSE 2014), 2014, pp. 88–95.
\textsuperscript{5} https://pais.hse.ru/research/projects/gena

\textsuperscript{5} https://pais.hse.ru/research/projects/gena