SYRCoSE 2014

Editors:

Alexander Kamkin, Alexander Petrenko and
Andrey Terekhov

Preliminary Proceedings of the 8th Spring/Summer Young Researchers' Colloquium on Software Engineering

Saint Petersburg, May 29-31, 2014

2014

Preliminary Proceedings of the 8th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2014), May 29-31, 2014 – Saint Petersburg, Russia:

The issue contains the papers presented at the 8th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2014) held in Saint Petersburg, Russia on May 29-31, 2014. Paper selection was based on a competitive peer review process being done by the program committee. Both regular and research-in-progress papers were considered acceptable for the colloquium.

The topics of the colloquium include formal methods, embedded system design, system programming, process mining, testing, compiler technologies and others.

Предварительный сборник трудов 8-ого весеннего/летнего коллоквиума молодых исследователей в области программной инженерии (SYRCoSE 2014), 29-31 мая 2014 г. – Санкт Петербург, Россия:

Сборник содержит статьи, представленные на 8-ом весеннем/летнем коллоквиуме молодых исследователей в области программной инженерии (SYRCoSE 2014), прошедшем в Санкт Петербурге 29-31 мая 2014 г. Отбор статей производился на основе рецензирования материалов программным комитетом. На коллоквиум допускались как полные статьи, так и краткие сообщения, описывающие текущие исследования.

Программа коллоквиума охватывает следующие темы: формальные методы, проектирование встроенных систем, системное программирование, анализ процессов, тестирование, компиляторные технологии и др.

ISBN 978-5-91474-020-4

Contents

Formal Methods

Embedded System Design and System Programming

Process Mining

Testing

Compiler Technologies

Application-Specific Methods and Tools

Foreword

Dear participants, we are glad to meet you at the 8th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE). The event is held in Saint Petersburg, the second largest city in Russia and its cultural capital. The colloquium is hosted by Saint Petersburg State Polytechnical University (SPbSPU), one of the top research and educational institutions in Russian Federation in the field of applied physics and mathematics, industrial engineering, chemical engineering, aerospace engineering and many other disciplines. SYRCoSE 2014 is organized by Institute for System Programming of the Russian Academy of Sciences (ISPRAS) and Saint Petersburg State University (SPbSU) jointly with SPbSPU.

In this year, Program Committee (consisting of 50 members from more than 25 organizations) has selected 31 papers. Each submitted paper has been reviewed independently by three referees. Participants of SYRCoSE 2014 represent well-known universities, research institutes and companies such as Aarhus University, Exactpro Systems, ISPRAS, JSS Research Foundation, JSS Academy of Technical Education, Kostroma State Technological University, National Research University – Higher School of Economics, Obninsk Institute for Nuclear Power Engineering, Perm State National Research University, Saint Petersburg Electrotechnical University "LETI", SPbSPU, SPbSU, TELECOM SudParis, Tomsk State University, Ulyanovsk State Technical University, Yandex, Yaroslavl State University and Yuri Gagarin State Technical University of Saratov (4 countries, 13 cities and 18 organizations).

We would like to thank all of the participants of SYRCoSE 2014 and their advisors for interesting papers. We are also very grateful to the PC members and the external referees for their hard work on reviewing the papers and selecting the program. Our thanks go to the invited speakers, Bertrand Meyer (ETH Zürich, Switzerland) and Kostya Serebryany (Google Moscow, Russia). We would also like to thank our sponsors and supporters: Russian Foundation for Basic Research (grant 14-07-06006), Google, Exactpro Systems and CyberLeninka. Finally, our special thanks to local organizers, Igor Chernorutskiy, Vsevolod Kotlyarov and Tatyana Elamic (SPbSPU), for their invaluable help in organizing the colloquium in Saint Petersburg.

Sincerely yours

Alexander Kamkin, Alexander Petrenko and Andrey Terekhov
May 2014

Committees

Program Committee Chairs

Alexander Petrenko – Russia
Institute for System Programming of RAS

Andrey Terekhov – Russia
Saint-Petersburg State University

Program Committee

Jean-Michel Adam – France
Pierre Mendès France University

Sergey Avdoshin – Russia
National Research University Higher School of Economics

Eduard Babkin – Russia
National Research University Higher School of Economics

Svetlana Chuprina – Russia
Perm State National Research University

Pavel Drobintsev – Russia
Saint-Petersburg State Polytechnic University

Liliya Emaletdinova – Russia
Institute for Technical Cybernetics and Informatics, KNRTU

Victor Gergel – Russia
Lobachevsky State University of Nizhny Novgorod

Efim Grinkrug – Russia
National Research University Higher School of Economics

Maxim Gromov – Russia
Tomsk State University

Vladimir Hahanov – Ukraine
Kharkov National University of Radioelectronics

Shihong Huang – USA
Florida Atlantic University

Iosif Itkin – Russia
Exactpro Systems

Alexander Kamkin – Russia
Institute for System Programming of RAS

Vsevolod Kotlyarov – Russia
Saint-Petersburg State Polytechnic University

Oleg Kozyrev – Russia
National Research University Higher School of Economics

Vladimir Kozyrev – Russia
National Research Nuclear University "MEPhI"

Daniel Kurushin – Russia
State National Research Polytechnic University of Perm

Peter Gorm Larsen – Denmark
Aarhus University

Rustam Latypov – Russia
Institute of Computer Science and Information Technologies, KFU

Alexander Letichevsky – Ukraine
Glushkov Institute of Cybernetics, NAS

Alexander Lipanov – Ukraine
Kharkov National University of Radioelectronics

Irina Lomazova – Russia
National Research University Higher School of Economics

Ludmila Lyadova – Russia
National Research University Higher School of Economics

Victor Malyshko – Russia
Moscow State University

Vladimir Makarov – Russia
Yaroslav-the-Wise Novgorod State University

Tiziana Margaria – Germany
University of Potsdam

Marek Miłosz – Poland
Institute of Computer Science, Lublin University of Technology

Alexey Namestnikov – Russia
Ulyanovsk State Technical University

Valery Nepomniaschy – Russia
Ershov Institute of Informatics Systems

Mykola Nikitchenko – Ukraine
Kyiv National Taras Shevchenko University

Yuri Okulovsky – Russia
Ural Federal University

Elena Pavlova – Russia
Microsoft Research

Ivan Piletski – Belorussia
Belarusian State University of Informatics and Radioelectronics

Vladimir Popov – Russia
Ural Federal University

Yury Rogozov – Russia
Taganrog Institute of Technology, Southern Federal University

Rustam Sabitov – Russia
Kazan National Research Technical University

Nikolay Shilov – Russia
Ershov Institute of Informatics Systems

Ruslan Smelyansky – Russia
Moscow State University

Valeriy Sokolov – Russia
Yaroslavl Demidov State University

Petr Sosnin – Russia
Ulyanovsk State Technical University

Veniamin Tarasov – Russia
Povolzhskiy State University of Telecommunications and Informatics

Sergey Ustinov – Russia
Saint-Petersburg State Polytechnic University

Vladimir Voevodin – Russia
Research Computing Center of Moscow State University

Dmitry Volkanov – Russia
Moscow State University

Mikhail Volkov – Russia
Ural Federal University

Nadezhda Yarushkina – Russia
Ulyanovsk State Technical University

Rostislav Yavorsky – Russia
National Research University Higher School of Economics

Nina Yevtushenko – Russia
Tomsk State University

Vladimir Zakharov – Russia
Moscow State University

Sergey Zaydullin – Russia
Kazan National Research Technical University

Organizing Committee Chairs and Secretaries

Igor Chernorutskiy – Russia
Saint-Petersburg State Polytechnic University

Tatyana Elamic – Russia
Saint-Petersburg State Polytechnic University

Alexander Kamkin – Russia
Institute for System Programming of RAS

Alexander Petrenko – Russia
Institute for System Programming of RAS

Vsevolod Kotlyarov – Russia
Saint-Petersburg State Polytechnic University

Referees

Jean-Michel Adam
Sergey Avdoshin
Eduard Babkin
Mikhail Chupilko
Svetlana Chuprina
Pavel Drobintsev
Denis Efremov
Victor Gergel
Efim Grinkrug
Maxim Gromov
Shihong Huang
Iosif Itkin
Dmitry Ivankov
Anna Kalenkova
Alexander Kamkin
Dmitry Kosolobov
Vsevolod Kotlyarov
Artem Kotsynyak
Vladimir Kozyrev
Daniel Kurushin
Peter Gorm Larsen
Alexander Lipanov
Irina Lomazova
Lyudmila Lyadova
Victor Malyshko
Tiziana Margaria
Ivan Mikhailov
Alexey Namestnikov
Mykola Nikitchenko
Yuri Okulovsky
Nikolay Pakulin
Elena Pavlova
Alexander Petrenko
Ivan Piletski
Vladimir Popov
Yury Rogozov
Nikolay Shilov
Sergey Smolov
Valeriy Sokolov
Petr Sosnin
Veniamin Tarasov
Andrei Tatarnikov
Andrey Terekhov
Dmitry Volkanov
Mikhail Volkov
Nadezhda Yarushkina
Rostislav Yavorskiy
Nina Yevtushenko
Vladimir Zakharov

Checking Conformance of High-Level Business Process Models to Event Logs

Antonina K. Begicheva
National Research University
Higher School of Economics (HSE)
33 Kirpichnaya Str., Moscow, Russia
Email: be-ton@yandex.ru

Irina A. Lomazova
National Research University
Higher School of Economics (HSE)
33 Kirpichnaya Str., Moscow, Russia
Email: ilomazova@hse.ru

I. Introduction

Process mining [1] is a new technology, providing a variety of methods to discover, monitor and improve real processes by extracting knowledge from event logs. The two most prominent process mining tasks are: (i) process discovery: constructing a process model from example behavior recorded in an event log, and (ii) conformance checking: diagnosing and quantifying discrepancies between observed behavior and modeled behavior. There are many software products which allow us to use methods of Process Mining. ProM [4] is an open-source tool supporting many techniques of Process Mining, which are represented as plug-ins. Due to a flexibility of this environment it can be used both for reserch and applications.

This paper studies conformance checking [1], [3], [6], [7]. Conformance checking uses both an event log and a model, and compares observed behavior written in the log with the behavior produced by the model. The general goal is to find discrepancies between them to improve a model. Conformance checking techniques can also be used for measuring the performance of process discovery algorithms (that restores a model on the basis of a known log) and to repair models that have no a well alignment with the real behavior of the process.

There are four model's evaluation criteria: fitness, precision, generalization and simplicity. Fitness measures ”the proportion of behavior in the event log possible according to the model”. Among the four quality criteria, fitness is the most related to the conformance. Several methods of conformance checking were developed. We consider methods based on *replay* approach [3]. Replaying a log on a model can help to measure *fitness*.

An obvious approach to measure fitness would be just to count the fraction of cases that can be ”parsed completely” (i.e. the proportion of cases corresponding to firing sequences leading from *[start]* to *[end]*). Fitness can range from 0 to 1. It is supposed that fitness is equal to 1, if the log *perfectly fits* the model. When measuring fitness by replaying, we could stop replaying a trace when we face a problem and mark this trace as unsuitable. We get more information about conformance if we continue replaying the trace on the model, and record a count of all missing tokens and all tokens that are pending at the end.

Let us denote the number of produced tokens by p, the number of consumed tokens by c, the number of missing tokens by m and the number of remaining tokens by r. Initially, when all places are empty, $p = c = 0$. Then the environment produces a token for the place $[start]$. Therefore, the p counter is incremented: $p \leftarrow p + 1$. A log is replayed by consecutive firings of transitions, corresponding to activities of the process. Each transition consumes and produces several tokens and we increase the corresponding variables. If we need an extra token in a place to continue replaying (when the next transition is not enabled), then the m counter must be incremented and the place, that lacks a token, is marked as a place where a token was missed. If by the time of consuming a token from the place *[end]* there were tokens pending in some other places, the r counter must be increased by the number of the remained tokens, and the places with tokens must be tagged.

The fitness of a trace σ of a workflow process model N is defined as follows:

$$fitness(\sigma, N) = \frac{1}{2}(1 - \frac{m}{c}) + \frac{1}{2}(1 - \frac{r}{p})$$

When working with business processes we typically use detailed logs, which present the full report about sequentially executed activities. Since in most information systems logs are generated automatically, keeping detailed records is not a problem. However, large and detailed models are not good to deal with. Such models are not clear and readable for experts. Experts prefer to work with more abstract (high-level) models. More abstract models are easier to construct, understand and analyze. Process models developed by people are, as a rule, not very large and abstract from technical details. So, checking conformance of an abstract model and a low-level event log, generated by an information system, is an important and challenging problem. However, as far as we know, this problem was not studied in the literature.

In this paper we consider an abstract model in which each separate activity represents a subprocess built from a set of smaller activities. A history of a detailed process behavior is recorded in low-level logs. Process models are represented by workflow nets — a special subclass of Petri nets [2]. We present a method for checking conformance of an abstract model and a low-level event log.

The paper is organized as follows. Section II contains some basic definitions and notions, including Petri nets, event log, perfect fits and refinement. In Section III we give a motivating example of handling a request for a compensation within airline in terms of Petri nets. In Section IV we present a method for checking conformance between an abstract model and a low-level log. We also give a justification of this method by proving its correctness in the case of perfect fitness. An implementation of our algorithm is described in Section V. Section VI contains some conclusions.

II. Preliminaries

We start with recalling some basic notions from the set theory. Let S be a set. By S^* we denote the set of all finite sequences (words) over S.

$S = S_1 \cup S_2 \cup \ldots \cup S_n$ is a *partition* of S iff $\forall i, j \in [1, n]$: $S_i \subseteq S$ and $S_i \cap S_j = \emptyset$.

A *multiset* m over a set S is a mapping $m : S \to Nat$, where *Nat* is the set of natural numbers (including zero), i.e. a multiset may contain several copies of the same element.

For two multisets m, m' we write $m \subseteq m'$ iff $\forall s \in S$: $m(s) \leq m'(s)$ (the inclusion relation). The sum of two multisets m and m' is defined as usual: $\forall s \in S : (m + m')(s) = m(s) + m'(s)$, the difference is a partial function: $\forall s \in S$ such that $m(s) \geq m(s') : (m - m')(s) = m(s) - m'(s)$. By $\mathcal{M}(S)$ we denote the set of all finite multisets over S. Non-negative integer vectors are often used to encode finite multisets.

Definition 1 (Petri net)**.** Let P and T be disjoint finite sets of *places* and *transitions* and $F\colon (P \times T) \cup (T \times P) \to Nat$. Then $N = (P, T, F)$ is a *Petri net*. Let A be a finite set of activities. A labeled Petri net is a Petri net with a labeling function $\lambda : T \to A \cup \{\epsilon\}$ which maps every transition to an activity (a transition label) from A, or a special label ϵ, corresponding to an invisible action.

A *marking* in a Petri net is a function $m\colon P \to Nat$, mapping each place to some natural number (possibly zero). Thus a marking may be considered as a multiset over the set of places. Pictorially, P-elements are represented by circles, T-elements by boxes, and the flow relation F by directed arcs. Places may carry tokens represented by filled circles. A current marking m is designated by putting $m(p)$ tokens into each place $p \in P$.

For a transition $t \in T$ an arc (x, t) is called an *input arc*, and an arc (t, x) — an *output arc*; the *preset* ${}^\bullet t$ and the *postset* $t^\bullet$ are defined as the multisets over P such that ${}^\bullet t(p) = F(p, t)$ and $t^\bullet(p) = F(t, p)$ for each $p \in P$.

A transition $t \in T$ is *enabled* in a marking m iff $\forall p \in P$ $m(p) \geq F(p, t)$. An enabled transition t may *fire* yielding a new marking $m' =_{\text{def}} m - {}^\bullet t + t^\bullet$, i. e. $m'(p) = m(p) - F(p, t) + F(t, p)$ for each $p \in P$ (denoted $m \xrightarrow{t} m'$, $m \xrightarrow{\lambda(t)} m'$, or just $m \to m'$).

A *Workflow-net* is a (labeled) Petri net with two special places: i and f. These places are used to mark the beginning and the ending of a workflow process.

Definition 2 (Workflow net)**.** A (labeled) Petri net $N = (P, T, F, \lambda)$ is called a *workflow net (WF-net)* iff

1) There is one source place $i \in P$ and one sink place $f \in P$ s. t. ${}^\bullet i = f^\bullet = \emptyset$;
2) Every node from $P \cup T$ is on a path from i to f.
3) The initial marking in N contains the only token in its source place.

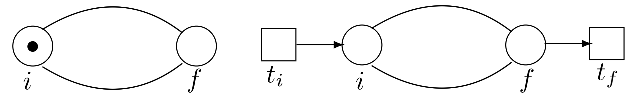

Fig. 1. Extending a WF net with initial and final transitions

By abuse of notation we denote by i both the source place and the initial marking in a WF-net. Similarly, we use f to denote the final marking in a WF-net N, defined as a marking containing the only token in the sink place f.

Let $N = (P, T, F, \lambda)$ be a WF-net. The *extended WF net (EWF-net)* $N' = (P', T', F', \lambda')$ is defined as follows: $P' = P, T' = T \cup \{t_i, t_f\}$, and $F' = F \cup \{\langle t_i, i\rangle, \langle f, t_f\rangle\}$, where t_i, t_f are new (not occurring in P, T) nodes. The new transitions t_i, t_f are labeled with invisible activity ϵ in N', all other transitions in N' have the same labels as in N. In the remainder we will denote such an extended WF net of N as $t^+(N)$. The initial marking in an extended WF net contains no tokens. Thus an extended WF net may start a new case at any moment (cf. Fig.1).

Event logs keep a history of process executions.

Definition 3 (Event log)**.** Let A be a finite set of activities. A *trace* σ is a finite sequence of activities, i.e., $\sigma \in A^*$. An *event log* L is a finite multiset of traces, i.e., $L \in \mathcal{M}(A^*)$.

In this paper we study conformance checking. Given a model and an event log we would like to compare the process model behavior and the behavior recorded in the event log. Several metrics for conformance checking were defined in the literature [1]. Among the most important metrics is *fitness*. Informally speaking, fitness measures the proportion of behavior in the event log possible according to the model.

Definition 4 (Perfect fit)**.** Let N be a WF-net with transition labels from A, an initial marking i, and a final marking f. Let σ be a trace over A. We say that a trace $\sigma = a_1, \ldots, a_k$ *perfectly fits* N iff there exists a sequence of firings $i = m_0 \xrightarrow{t_1} \ldots \xrightarrow{t_k} m_{k+1} = f$ in N, s.t. the sequence of activities $\lambda(t_1), \lambda(t_2), \ldots, \lambda(t_k)$ after deleting all invisible activities ϵ coincides with σ. A log L *perfectly fits* N iff every trace from L perfectly fits N.

Petri nets can be extended with hierarchy and it is done e.g. in Colored Petri nets (CPN) [8]. Hierarchy allows to develop more compact models with a compositional network structure. In the case of two-level hierarchy there are two models of one process: a high-level (*abstract*) model and a low-level (*refined*) model. The high-level model is a model with abstract transitions. An abstract transition refers to a Petri net subprocess model refining the activity represented by this transition. The low-level model can be obtained from an abstract model by substituting subprocess models for abstract transitions.

Definition 5 (Substitution)**.** Let $N_1 = (P_1, T_1, F_1, \lambda_1)$ be a WF-net, $t \in T$ be a transition in N_1. Let also $N_2 =$

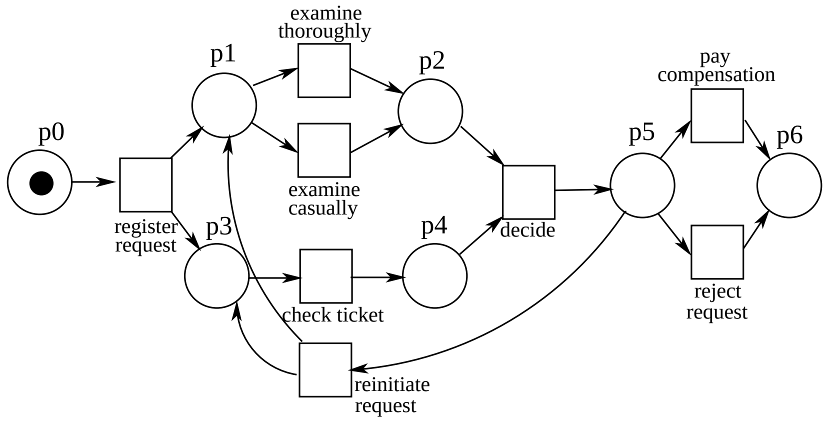

Fig. 2. An abstract model for handling compensation requests

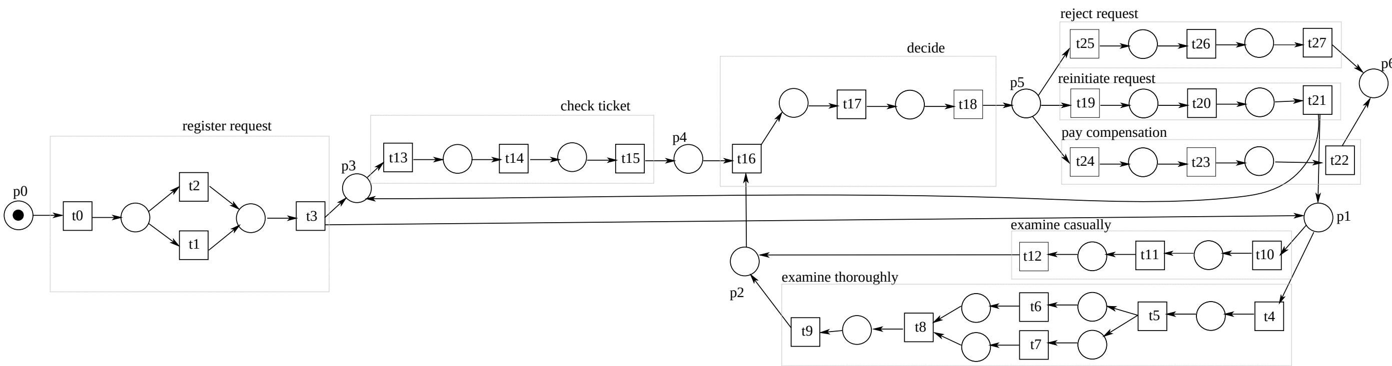

Fig. 3. A refined model for handling compensation requests, which refines the model in Fig. 2

$(P_2, T_2, F_2, \lambda_2)$ be an EWF-net with the initial and final transitions t_i, t_f correspondingly. We say that a WF-net $N_3 = (P_3, T_3, F_3, \lambda)$ is obtained by a *substitution* $[t \to N_2]$ of N_2 for t in N_1 iff $P_3 = P_1 \cup P_2$, $T_3 = T_1 \cup T_2 \setminus \{t\}$, $F_3 = F_1 \cup F_2 \setminus \{(p,t) \mid p \in {}^\bullet t\} \setminus \{(t,p) \mid p \in t^\bullet\} \cup \{(p,t_i) \mid p \in {}^\bullet t\} \cup \{(t_f, p) \mid p \in t^\bullet\}$,

Definition 6 (Refinement)**.** Let N_a, N_r be two WF-nets with sets of activities A_a, A_r correspondingly. Let $A_a = a_1, a_2, \ldots, a_n$, and $A_r = A_r^1 \cup A_r^2 \cup \ldots \cup A_r^n$ be a partition of A_r into n subsets, and $N^1, N^2, \ldots N^n$ be EWF-nets with sets of activities $A_r^1, \ldots, A_r^n$ correspondingly. We say that N_r is a refinement of N_a via substitutions $[a_1 \to N_r^1, a_2 \to N_r^2, \ldots a_n \to N_r^n]$ iff N_r can be obtained from N_a by simultaneous substitutions of N_r^i for all t s.t. $\lambda(t) = a_i$.

III. Motivating Example

Let us consider a toy model from [1], which describes handling a request for a compensation within airline. Here customers may request compensations for various reasons. An abstract model of this process (expressed in terms of a Petri net) is presented in Fig.2. Fig.3 presents a refined model of the same process. To avoid congestion of activities' names in the low-level model in Fig.3 only places inherited from the abstract model are labeled in the picture.

Let us explain the correspondence between the two models. Let $N^0, N^1, N^2, N^3, N^4, N^5, N^6, N^7$ be EWF-nets with sets of activities $A^0 = \{t0, t1, t2, t3\}, A^1 = \{t4, t5, t6, t7, t8, t9\}, A^2 = \{t10, t11, t12\}, A^3 = \{t13, t14, t15\}, A^4 = \{t16, t17, t18, \}, A^5 = \{t19, t20, t21\}, A^6 = \{t22, t23, t24\}, A^7 = \{t25, t26, t27\}$ correspondingly. The refined model in Fig. 3 is the refinement of the abstract model in Fig. 2 via the substitutions $[register\_request \to N^1, examine\_thoroughly \to N^2, examine\_casually \to N^3, check\_ticket \to N^3, decide \to N^4, reinitiate\_request \to N^5, pay\_compensation \to N^6, reject\_request \to N^7]$. A sample of an event log obtained for the refined model L_r is shown in Fig.4. Note that for the log L_r (Fig.4) and the refined model (Fig.3) we have $fitness = 1$, since the log L_r is generated by the model.

So, we have an abstract model (as a more simple to understand and analyse) and we want to check conformance of this model to a low-level log. It is obvious that we cannot do it straightforward, since the model is defined in terms of abstract activities, and the log contains low-level activities.

IV. Checking Conformance between an Abstract Model and a Refined Event Log

To check conformance between an abstract model and a low-level log, we first transform the given log into a log over abstract activities. For this purpose, each low-level activity in the log is replaced by a name of the subprocess (an abstract

$$
\begin{aligned}
L = \{ & < t0, t1, t3, t4, t5, t6, t13, t14, t7, t8, t15, t9, t16, t17, t18, t25, t26, t27 >, \\
& < t0, t1, t3, t4, t13, t14, t5, t7, t15, t6, t8, t9, t16, t17, t18, t24, t23, t22 >, \\
& < t0, t1, t3, t13, t10, t11, t14, t15, t12, t16, t17, t18, t24, t23, t22 >, \\
& < t0, t2, t3, t13, t4, t14, t15, t5, t6, t7, t8, t9, t16, t17, t18, t25, t26, t27 >, \\
& < t0, t2, t3, t4, t13, t14, t15, t5, t7, t6, t8, t9, t16, t17, t18, t24, t23, t22 >, \\
& < t0, t1, t3, t10, t11, t13, t12, t14, t15, t16, t17, t18, t19, t20, t21, t10, t11, t13, t14, t15, t12, t16, t17, t18, t25, t26, t27 >, \\
& < t0, t2, t3, t13, t10, t14, t15, t11, t12, t16, t17, t18, t24, t23, t22 >, \\
& < t0, t2, t3, t13, t10, t11, t12, t14, t15, t16, t17, t18, t19, t20, t21, t10, t13, t14, t11, t15, t12, t16, t17, t18, t24, t23, t22 >, \\
& < t0, t2, t3, t13, t14, t10, t11, t15, t12, t16, t17, t18, t25, t26, t27 >\}.
\end{aligned}
$$

Fig. 4. An event log for the refined model in Fig. 3

activity) it belongs to. Hence we get a log with "stuttering" abstract activities. This transformation is implemented by the method *toHighLevel()*, schematically presented in Algorithm 1.

Data: *lowlevellog* — a list of low-level activities, *hlaction* — a set of high-level activities, where for each high-level activity is stored information about its partition into subsets of low-level activities.
Result: *highlevellog* — a high-level event log.
$i \longleftarrow 0$;
$highlevellog \longleftarrow \emptyset$;
$currentLowAction \longleftarrow lowlevellog[i]$ **while** *i* < *lowlevellog.size* **do**
    `// search of high-level activity,`
    `// subsets of which contains this`
    `// low-level activity`
    $currentHighAction \longleftarrow search(hlaction, currentLowAction)$;
    if $currentHighAction \neq \emptyset$ **then**
        `// check of condition that`
        `// low-level activity is included`
        `// to partition of the current`
        `// high-level action`
        while *i* < *lowlevellog.size* **and** *currentHighAction.contains(currentLowAction)* **do**
            $i \longleftarrow i + 1$;
        end
        $highlevellog.add(currentHighAction)$;
    end
end
return $highlevellog$;

Algorithm 1: Method *toHighLevel()*, transforming a low-level log into a log over abstract activities

After converting the refined log into notations of the abstract model we get a new log, which is a multiset of sequences of abstract activities. But this still can not be used for the conformance checking because of stuttering actions. Moreover, when we have two concurrent subprocesses, represented by two concurrent abstract activities in an abstract model, stuttering sequences may interleave. To overcome this problem we transform an abstract model into a model allowing stuttering of each abstract activity. For this purpose we add

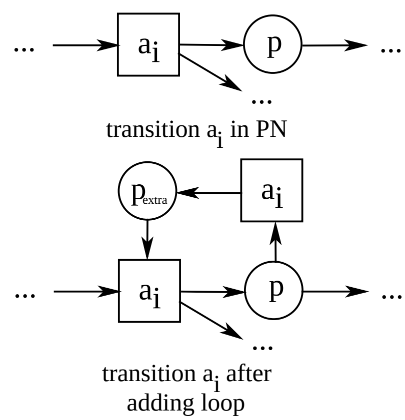

Fig. 5. Extending a transition by adding loop.

loops to transitions in the abstract model.

Algorithm 2 schematically describes the method *addLoops()* for transforming an abstract model by adding loops to abstract transitions (cf . Fig.5).

Now we describe the general algorithm of conformance checking between and abstract model and a low-level log in more details.

Main Algorithm *(Converting a refined log to an abstract one and transforming the high-level model for working with result of this conversion).*
Let $N_a = (P, T, F, \lambda)$ be a Petri-net corresponding to an abstract model of a process over a set of activities A_a. Let also L_r be a low-level event log (a finite multiset of traces) over a set A_r of low-level activities.

Let A_a, A_r be the set of activities in the abstract model and a set of activities in a refined model correspondingly. Denote by and denote trace from it by $\sigma_r^j \in L_r$ where j is a number of trace.

1) Convert L_r to a high-level event log (denote it L_a) using data about the partition of A_r. Replace each $a_k \in \sigma_r^j$ (where k is a number of activities in the trace) to the corresponding activity from A_a for all j and k.
2) Use a rule introduced by us about repetitive activities in L_a. Replace every sequences of identical activities

```
Data: Petri net as petrinet, trace from event log as trace
Result: modified Petri net
trace ⟵ sort(trace);
// index of current activity
index ⟵ 0;
currentActivity ⟵ trace[index];
while index < trace.size - 1 do
    indexOfNext ⟵ index + 1;
    nextActivity ⟵ trace[indexOfNext];
    if currentActivity==nextActivity then
        // check count of transition
        // with this name in Petri net
        if countInNet(petrinet, currentActivity) == 1
        then
            // add loop to Petri net
            // for current activity
            addLoop(petrinet, currentActivity);
        end
        indexOfNext ⟵ indexOfNext + 1;
        while indexOfNextActivity < trace.size and
        currentActivity=nextActivity do
            indexOfNext ⟵ indexOfNext + 1;
        end
        index ⟵ indexOfNext − 1;
        currentActivity ⟵ trace[index] ;
    end
    return petrinet;
end
```

Algorithm 2: Method *addLoops()* for model's transformation by adding loops

in σ_r^j (for all j) to one instance of this activity, i.e. $[\{a_a^k \dots a_a^k\} \to a_a^k]$.

3) If the result obtained in the previous steps (L_a) does not contain traces with repetitive activities (unlike the previous step, they are non-consecutive like $\{a_r^1, \dots, a_a^k, a_a^{k+1}, \dots, a_a^k, \dots, a_a^n\}$), then stop, otherwise proceed to the next step.
4) Working with each trace individually find all a_a, which have repeats in the same trace (see the previous step) and transitions in N_a, which correspond to these actions.
5) Add a loop to N_a for all transitions from the previous step (denote the current transition by t):
    a) Choose one place among $p_i \in P$ and $p_i \in t^\bullet$ (denote it by p').
    b) Add to N_a a new transition (denote it by t').
    c) Add to N_a a new place (denote it by p'').
    d) Add to N_a a new arcs: $\{(p', t'); (t', p''), (p'', t)\}$
6) Apply any known algorithm for conformance checking of L_a to N_a.

We illustrate the algorithm by applying it to the example that was presented above in Fig. 2 and 4.

First, we convert the event log L_r to a high-level log by applying Algorithm 1. The log obtained as the result of this is denoted by L_a and is shown in Fig. 6. Then we apply Algorithm 2 to the abstract model N_a and obtain the new model N_a', shown in Fig. 7. The model N_a' is a stuttering model over abstract activities. And finally we check conformance between the model N_a' and the log L_a by replaying traces from L_a in N_a'. It turns out, that all traces from L_a can be replayed in N_a', i.e. the log L_a perfectly fits N_a'. This is not by chance. The following theorem states, that the proposed conformance checking method is stable under perfect fitness.

Theorem 1. Let N_r be a refinement of N_a and L_r be an event log over the set of activities A_r, i.e. $L_r \in \mathcal{M}(A_r^*)$. If L_r perfectly fits N_r, then the main algorithm return 1, which is interpreted as L_r perfectly fits N_a.

We omit the proof of the theorem, since it is rather technical and straightforward.

V. Implementation

The proposed method for checking conformance of high-level business model to low-level event log is implemented as a plug-in for ProM.

Our tool consists of six main classes:

1) *TransformerForConformanceChecking* class is responsible for interaction with framework and GUI.
2) *HighLevelTransition* class represents a high-level transition. Each object of this type have a name of appropriate abstract activity and an array of low-level activities, corresponding to this object.
3) *Activity* class represents an activity and implements forming of activity with data from event log.
4) *ConvertorForLowLevelLog* class is responsible for implementation of Algorithm 1, i.e. it transforms a low-level event log to an abstract event log.
5) *ConvertorForModel* class is responsible for implementation of Algorithm 2, i.e. it transforms an abstract model by adding the requisite loops.

VI. Conclusion

Abstract models are much more clear and more easily understood than low-level models. But in practice we have only low-level logs, which cannot be used for direct conformance checking. Hence checking conformance of a high-level business model to a low-level event log is an important task to facilitate the expert's work. In this paper we have presented a method for solving this problem. Also we had developed a ProM plug-in which implements the proposed algorithm.

We have proved, that our method recognizes perfect fitness between an abstract model and a low-level log correctly. This can be considered as a justification of the proposed approach. However, this is not enough. It is very important to check the method on logs with deviations. In the further research we plan experiments with different logs (logs with noise and different kinds of deviations), as well as real application logs, and we shall work on improving the algorithms through the use of found heuristics.

Acknowledgment

This study was carried out within the National Research University Higher School of Economics' Academic Fund.

$$
\begin{aligned}
L = \{ & < register\_request, examine\_thoroughly, check\_ticket, examine\_thoroughly, check\_ticket, \\
& examine\_thoroughly, decide, reject\_request >, \\
& < register\_request, examine\_thoroughly, check\_ticket, examine\_thoroughly, check\_ticket, \\
& examine\_thoroughly, decide, pay\_compensation >, \\
& < register\_request, check\_ticket, examine\_casually, check\_ticket, examine\_casually, decide, pay\_compensation >, \\
& < register\_request, check\_ticket, examine\_thoroughly, check\_ticket, examine\_thoroughly, \\
& decide, reject\_request >, \\
& < register\_request, examine\_thoroughly, check\_ticket, examine\_thoroughly, decide, pay\_compensation >, \\
& < register\_request, examine\_casually, check\_ticket, examine\_casually, check\_ticket, \\
& decide, reinitiate\_request, examine\_casually, check\_ticket, examine\_casually, decide, \\
& reject\_request >, \\
& < register\_request, check\_ticket, examine\_casually, check\_ticket, examine\_casually, \\
& decide, pay\_compensation >, \\
& < register\_request, check\_ticket, examine\_casually, check\_ticket, decide, reinitiate\_request, \\
& examine\_casually, check\_ticket, examine\_casually, check\_ticket, examine\_casually, decide, pay\_compensation >, \\
& < register\_request, check\_ticket, examine\_casually, check\_ticket, examine\_casually, \\
& decide, reject\_request > \}.
\end{aligned}
$$

Fig. 6. The abstract event log obtained by applying Algorithm 1 to the initial event log in Fig. 4

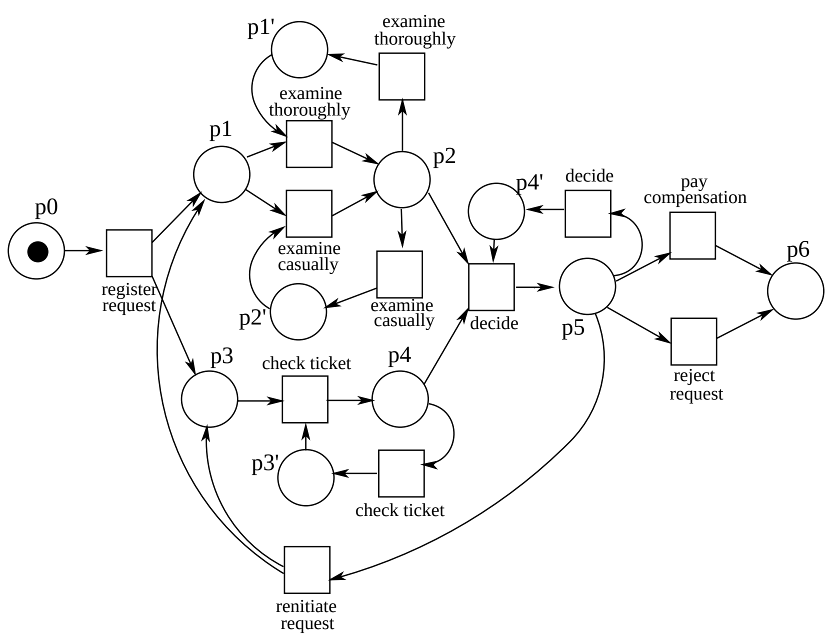

Fig. 7. The abstract model after adding loops (by applying Algorithm 2 to the model in Fig. 2)

References

[1] W.M.P. van der Aalst. *Discovery, Conformance and Enhancement of Business Processes.* Springer-Verlag, Berlin, 2011.

[2] W.M.P. van der Aalst, K.M. van Hee. *Workflow Management: Models, Methods and Systems*. MIT Press, 2002.

[3] A. Rozinat, and W.M.P. van der Aalst. Conformance Testing: Measuring the Alignment Between Event Logs and Process Models. *BETA Working Paper Series*, WP 144, Eindhoven University of Technology, Eindhoven, 2005.

[4] B.F. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM framework: A New Era in Process Mining Tool Support. Vol. 3536 of *Lecture Notes in Computer Science*, pp. 444-454, Springer, 2005.

[5] 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. 3439, 2010.

[6] A. Rozinat. Process mining: conformance and extension. *TU Eindhoven, Diss*, Eindhoven, 2010.

[7] A. Adriansyah, B. F. van Dongen, and W. M. P. van der Aalst. Conformance Checking Using Cost-Based Fitness Analysis. *IEEE 15th International Enterprise Distributed Object Computing Conference*, pp. 55-64, 2011.

[8] K. Jensen, and L. M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.