Iskra: A Tool for Process Model Repair

I. Shugurov <shugurov94@gmail.com>,
A. Mitsyuk <amitsyuk@hse.ru>,
Laboratory of Process-Aware Information Systems, National Research University Higher School of Economics, 3 Kochnovsky Proezd, Moscow, Russia

Abstract. This paper is dedicated to a tool whose aim is to facilitate process mining experiments and evaluation of the repair algorithms. Process mining is a scientific area which provides solutions and algorithms for discovery and analysis of business processes based on event logs. Process mining has three main areas of interest: model discovery, conformance checking and enhancement. The paper focuses exclusively on the tasks of enhancement. The goal of the enhancement process is to refine existing process models in order to make them conform to given event logs. The particular approach of enhancement, which is considered in the paper, is called decomposed model repair. It is assumed that event logs describe correct and up-to-date behavior of business processes, whereas process models may be erroneous. The proposed approach consists of several independent modules implementing different stages of the repair process. This architecture allows for more flexible repair configuration. Moreover, it allows researchers to conduct experiments with algorithms used by each module in isolation from other modules. Although the paper is devoted to the implementation part of the approach, theoretical preliminaries essential for domain understanding are provided. Moreover, a typical use case of the tool is shown as well as guides to extending the tool and enriching it with extra algorithms and functionality. Finally, other solutions which can be used for implementation of repair schemes are considered, pros and cons of using them are mentioned.

Keywords: Process model, Petri net, Model repair, Process mining.

DOI: 10.15514/ISPRAS-2015-27(3)-16

For citation: Shugurov I., Mitsyuk A. Iskra: A Tool for Process Model Repair. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 3, 2015, pp. 237-254. DOI: 10.15514/ISPRAS-2015-27(3)-16.

1. Introduction

In this paper, a tool for modular process model repair is presented. Architectural features and usage examples are provided.

Process mining [1] is a research area which deals with analysis of information systems or business processes by studying corresponding event logs and building

process models. The basic idea is that there can be significant improvements of existing systems, business operations if event logs and their content are studied more thoroughly. Three main aims of process mining are *process discovery*, *conformance checking* and *enhancement* [2].

The goal of *Process discovery* is to create a process model, based on an event log. That constructed model has to adequately describe the behavior observed in the event logs. The process of construction is typically called mining. As a model it is possible to use, for example, Petri nets. The challenge of process discovery is the hard truth that event logs reflect only a fraction of the overall process. It means that there may be activities, events, conditions, decision forks which exist in the initial process model, but they are not seen in event logs. For example, rare events in processes such as activities undertaken in emergency situations at nuclear power stations. Such activities exist, they are strictly regulated by rules and legislation, they influence the overall process a lot, but they are extremely uncommon so if an event log of a nuclear station is considered they are likely not to be present. Another serious issue concerning event logs is errors in them. Some events may be not put down in logs, log records might contain incorrect information about actually occurred events (i.e. wrong time stamp, event name) or they might be deliberately distorted.

Conformance checking is aimed to check whether a model fits a given event log. Because of the reasons presented in the description of process discovery, perfect fitness is almost not feasible in practice. Therefore, when discrepancy between a model and corresponding event logs occurs, it is desired to assess the significance of the deviation and highlight model parts where deviations take place [3, 4]. Some types of conformance checking algorithms support assigning weights to skipping and adding of events, that somehow indicates the significance of these deviations.

The reason for applying *Enhancement* is to improve already existing models by using information stored in event logs. Hence, the task here is to alter model, not to create an absolutely new one. Typical input parameters for the enhancement algorithms are a model and a corresponding event log. According to the presented definition, the approach the tool implements is categorized as an enhancement approach.

The remainder of this work is organized as follows.

In section *Process model repair* basic ideas behind model repair are described.

Modular repair approach section explains what modular repair is and how tools implementing this approach should be organized in order to achieve the goals.

In section *Tool overview* a summary of the tool functionality is reported.

Section *Tool architecture* contains information on the framework used during the development process, domain analysis and the architecture of the tool.

Section *Use case* shows step-by-step usage of the tool. In section *Related work* other tools are considered.

Section *Conclusion* concludes the paper.

2. Process model repair

In the field of process modeling not all the processes are of best quality. Usually process models are made by experts or obtained as a result of using automated model construction algorithms. In the field of process mining a lot of methods have been developed to discover models from process logs [1]. Real-life processes in information systems are complex and permanently changing. Thus, it is a very hard problem for experts and engineers to keep process models up to date.

The goal of model repair is to improve the quality of a model. In this paper, fitness is understood as a metric of model quality. Fitness is measured using technique described in [3]. Model repair has been introduced in [5]. As input for model repair a process model *M* and an event log *L* are used. If model *M* conforms to *L*, then there is no need to change *M*. If *M* partially does not conform to *L*, repair method has to repair non-conforming parts of *M*. Conforming parts of the model are kept as is. The result is the repaired model *M'*.

3. Modular repair approach

The implementation of the modular repair approach is the foundational goal of this work. The key idea is to make a general model repair scheme which will consist of several cells connected with strong links. A *cell* is understood as a placeholder where a user can put one of the appropriate algorithms. Cells are of the following types: (1) conformance checking cell, (2) decomposition cell, (3) selection cell, (4) repair cell, (5) composition cell, (6) final evaluation cell. Each cell type corresponds to the step in the modular repair.

Conformance checking cell is used to evaluate current progress of the repair process and indicate whether a current model quality is sufficient. An algorithm in a *decomposition cell*, as it is clear from its name, is responsible for dividing an entire model into smaller parts, which are easier to understand, analyze and repair. Decomposition for process mining is described in [6]. A *selection cell* includes an algorithm whose aim is to run conformance checking for each model part and decide which of them are sufficiently fit. A *repair cell* can be either a process discovery algorithm or some enhancement algorithm; although for generalization reasons they are called repairers in the paper. Once the decomposed parts are repaired they ought to be merged in order to form a single model. It is done by an algorithm located in a *composition cell*. An algorithm located in a *final evaluation cell* is executed after completion of the entire repair task. At this step several metrics are measured in order to assess the quality of the model and the repair. Moreover, similarity of the initial and the final models is checked. In the future, visualization of model differences will be incorporated.

At the first step the tool checks whether a model and a log conform to each other. The second step is one of the model decomposition methods, which allows for splitting the model into parts [7]. At the third step the tool selects conforming and non-conforming parts by application of conformance checking method to each part,

obtained at the second step with the projection of the event log onto set of activities corresponding to this part. The fourth step is the repair step. At this step the tool applies a repair algorithm. It can be, for example, simple re-discovery algorithm. By applying it the tool obtains a new model from the log part corresponding to a non-conforming part of the initial model. At the fifth step the tool composes all parts of the model into the final model using an appropriate method. The sixth step is the final evaluation of the repaired model. Usually, each algorithm has to be wrapped in additional code in order to be embedded in a particular cell of the tool.

This work will not consider the aspects of the methods which can be placed into cells. There is a theory behind each step of the repair process. Methods offer a lot of settings and options. Because of that, it will be impossible to put all the details in one text. The main goal of this work is to propose a software architecture that allows for exploring different algorithms and their features in the context of model repair.

4. Tool overview

The main functionality provided by the tool implies the following aspects:

- The tool allows users to select a decomposition method which, in their opinion, is the most suitable for a given model.
- The tool makes it possible to choose a repair algorithm. The choice of the algorithm is typically based on the properties of the algorithm and a model it produces. The task of choosing the best repair algorithm is basically an attempt to find appropriate alternative between time needed for the algorithm to do its work, presence or absence of so-called silent transitions (i.e. transitions that do not correspond to any events observed in an event log, but considered to be present because they somehow explain the model behavior) and conformance between a given model and an event log.
- One may specify importance of each metric for a particular repair task. This step is essential for automatic evaluation of how well the tool helps researchers achieve the desired repair result.
- Numeric results of the final model evaluation can be stored in CSV file either manually or automatically. CSV files are chosen because a lot of tools support this format, therefore, it significantly simplifies the further analysis or visualization. The evaluation process assesses the following metrics: fitness (two approaches for fitness measurement are employed), conformance, complexity and a similarity coefficient.
- The tool is responsible for visualization of each step the tool performs and a final model. In the future, the tool will also be fitted with a convenient visualization of the difference between an initial and a final model.
- The tool makes it possible to significantly modify logic the cells use, thus extending the tool or adjusting it to a particular circumstance.

It goes without saying that despite the existence of some theoretical guidelines, choosing the right decomposition and repairing algorithms as well as their settings can be extremely complicated and mean, in the worst-case scenario, brute-force seeking the right methods. Because of that, one of the tool's aims is to facilitate this very tedious process. Moreover, if one is developing or evaluating a repair algorithm, it will imply a lot of repetitive executions of it. Hence, the tool facilitates this process a lot and is likely to significantly reduce time spent on such tasks.

4. Tool overview

4.1 ProM

The tool is being developed using Java 6 Standard Edition and ProM 6.4 Framework [8]. ProM 6.4 is an open source framework specially designated for implementing process mining algorithms in a standardized way. ProM 6.4 consists of the core part and disjoints parts called plugins. The core part of the framework is responsible only for uploading available plugins, handling plugins' life cycle, interaction between plugins and basic functions for dealing with graphical user interface. Hence, programmers focus exclusively on implementation of algorithms, working with file system and visualization of results. The framework is distributed under GNU Public License, although some plugins are distributed under different licenses.

Once a plugin is properly configured, ProM automatically finds and uploads it, then this plugin is added to the list of all available plugins. In addition, the list of plugins demonstrates parameters required by each plugin. By doing this, the framework simplifies providing parameters needed for plugins. Nowadays, almost all data types for working with Petri nets have been implemented and supplied with visualizers, so researchers and developers are eliminated of necessity to implement them from scratch.

Each plugin has so-called context. Context acts as a bridge between plugin and the framework because it is the only way plugins can interact and collaborate with ProM. For every context child contexts may be created, each of which is designated for a specific task. Thus, it is possible to construct a hierarchy of plugin calls from a parent plugin.

Plugins may run either with or without graphical user interface. The former provides a rich possibility to interact with user or visualize data, whereas the later enables to call other plugins in the background simultaneously with user interaction in the main plugin. ProM encourages developers to write extendable and loose coupled software, providing a rich set of tools. One of such tools, extensively used in the tool, is a mechanism for finding all classes annotated in a special way. Arguably the most common way to use annotations is to mark Java classes that contain algorithms. One creates an interface for a set of related algorithms, and then annotates each of them. After that, they can be easily found and used via annotations.

Interaction between plugins is accomplished by using a plugin manager. The plugin manager provides API for invoking plugins, makes sure that correct context is configured for a called plugin. The plugin manager enables not only to invoke known plugins but also to look for plugins with specific signature, to invoke them and to obtain results of their executions. Despite its promising flexibility and convenience, in practice it is generally easier to use conventional method invocations, because the API exposed by the plugin manager is a bit unintuitive. Furthermore, direct methods calls ensure more readable code. Because of these reasons, the direct methods call are preferred in the tool and used wherever possible.

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

The tool, which implements the approach presented in this work, is built as a plugin for the ProM Framework; therefore architecture of the tool has to fulfill all the aforementioned requirements for ProM plugins. We decided to use such an approach because the framework already has plugins which take care of discovery of Petri nets, event logs import and export, conformance checking as well as decomposition plugins, and provides further opportunities to work with the resulting data.

4.2 Preliminary domain analysis

This section is completely devoted to the analysis of the existing plugins for decomposition and model repair, because their usage involved extensive and from time to time tricky interaction with ProM and ProM plugin manager. In addition, the way how decomposition and repair model plugins are used is of high importance because it influences whether the tool is easy to extend. Detailed explanation of how conformance checking, final evaluation and the overall infrastructure are made is left to the following subsection.

The core implementation task of this project was to incorporate a dozen of available plugins for model repairing, decomposition and conformance checking, that have different authors, coding styles and settings. One of the main requirements for the resulting architecture was to make it as straightforward and comprehensive as possible, though ensure that it is flexible. In addition, we wanted to reuse as much of the existing code as possible. It meant that before the development of the tool could be started there was a need to scrutinize source code files of existing projects which we intended to use. This analysis was focused on 3 most important questions: (1) Does the architecture of each plugin follow MVC pattern [9]? (2) How heavily does each plugin use ProM-specific classes, tools? For example, can it be easily retrieved from ProM and used as some sort of standalone application? Do any of plugin show graphical user interface? (3) What set of parameters is required for each plugin?

The conducted analysis of repair algorithms revealed that the source code had been written in inconsistent way, the majority of plugins do not follow the MVC principles, which increased efforts needed for using them. As a result, plugins we intended to use were separated into 3 groups according to their coupling with ProM and the simplicity of their reuse:

- Plugins whose execution needs requesting via the plugin manager of ProM. Hence, in order to call them we supply plugin name, a list of required parameters and types of expected output. Then the plugin manager seeks the requested plugin and executes it. Examples of such plugins are *Alpha miner* [10] and *ILP Miner* [11].
- Plugins whose execution can be initiated via usual Java method calls without need to delegate this task to the ProM plugin manager. *Genetic miner* [12] and *Heuristics miner* [13] are placed in this group of plugins.
- Plugins whose architecture follows the MVC pattern. They are characterized by clear separation of actual algorithm and ProM-specific parts. Such plugins are more desirable because their usage and extension requires less time and effort. Unfortunately, *Inductive miner* [14] is the only plugin which falls into this category.

The subsequent step was to determine the ways which would allow users of the tool to specify parameters for repair algorithms if users wish to do it, otherwise default parameters would have to be set. The study of the plugins showed that only *Alpha miner* does not show GUI, whereas others do but have only one screen with settings, which allows for significant simplification of the resulting design decisions.

The situation with decomposition algorithms is a bit easier despite some nuances. First of all, they are highly sensitive to the input data. Event logs may include a lot of information in order to simplify further log analysis and error detection. ProM plugins responsible for projection a net on a log are aware of this information and try to make full use of it while projecting a net. By *projection* in this paper the process of extracting events which correspond to a particular model part from the entire event log is understood. Despite its high purpose, it is prone to produce rather unexpected outcome. It seems they work better and give correct result if event logs contain information only about event names. Concerning this issue it is absolutely essential to apply some kind of model and event log preprocessing techniques before trying to decompose and project a model. Furthermore, model decomposition is typically not a one-step process – it requires a number of consequent plugin calls, but for the sake of simplicity, covering up this circumstance from the main logic of our tool was on the list of the goals.

On the other hand, all decomposer plugins may be executed without showing GUI. In fact, only SESE decomposer [15] has one. Nevertheless, the possibility of existence of GUI was considered thoroughly due to extendibility and flexibility matters.

4.3 Usage of decomposition and repair algorithms

Judging by the results of the analysis of repair plugins we came up with a detailed plan on how to abstract from specific implementation details and provide a common interface for using these plugins. Of course, each of 3 plugin types (model repairing, model decomposition and conformance checking) has its own interface, unique for its specific nature. So, the final decision was to write "wrapper" interfaces and classes for required plugins. *Wrapper* is understood as a class which defines a common interface and hides the details how actual plugin is invoked. In fact, the concept of the adapter pattern [16] was exploited. The tool works only with such wrappers without any knowledge how inter-plugin communication is carried out. Furthermore, wrappers apply an idea of using annotations, which allows for complete deliverance from dependencies of the tool on wrappers and, hence, on external plugins. This approach also facilitates extension of the tool: those who are willing to incorporate new algorithms do not need gaining access to the source code of the tool. The only thing that has to be done is to create a Java class that extends either *IskraDecomposer* or *IskraRepairer* and marked by the corresponding annotation (either *@IskraDecomposer* or *@IskraRepairer*). Then ProM will detect this class and our tool will add it to the list of available algorithms. One important constraint is that wrappers must have an empty constructor. If a wrapper does not have it, the wrapper will not be available.

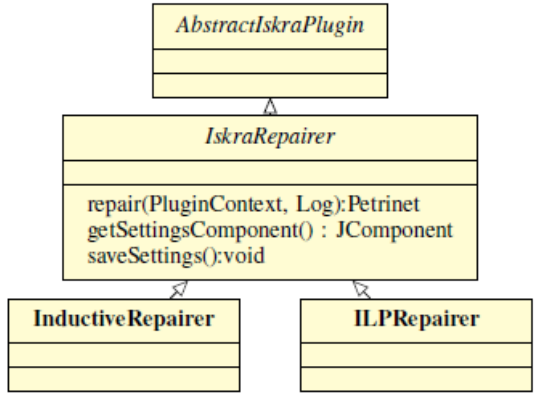

Fig. 1. Repairers hierarchy

Figure 1 and figure 2 depict the design of repairers and decomposers. Class *AbstractIskraPlugin* is a common superclass for all implemented wrappers. It encapsulates plugin's name and indicates that it is a plugin after all. Then, there are two abstract classes *IskraRepairer* and *IskraDecomposer* which provide a common interface respectively for repairers and decomposers. The tool uses only links to these classes, not to their subclasses. The architecture has been implemented and

proved to be viable. *InductiveRepairer*, *ILPRepairer*, *SESEDecomposer*, *PassageDecomposer* [17] are examples of actual (not abstract) classes. In order to save space and make a picture more comprehensible only these classes are shown, however half a dozen of others adhere to the architecture and available in the tool.

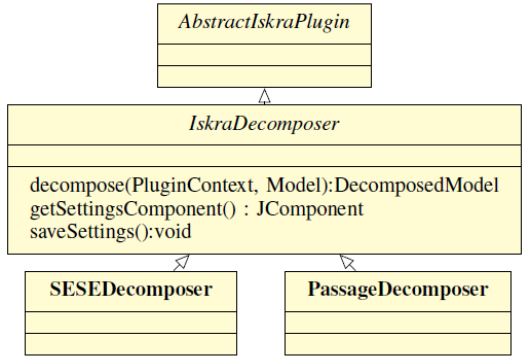

Fig. 2. Decomposers hierarchy

The typical scenario of using wrappers is:

1. Method *getSettingsComponent* is invoked.
2. If the value returned after the invocation *getSettingsComponent* is not *null*, then received GUI is displayed to a user.
3. GUI demonstration means having to save setting by invoking *saveSettings* method.
4. At this point a plugin is properly configured and is ready to be used. Only one thing left to get result – to invoke either *repair* or *decompose*.

It must be mentioned that steps 1-3 are arbitrary. If a user is either satisfied with default setting or does not want to show GUI then according to the contract, a wrapper supplies defaults settings to a corresponding plugin. If a plugin does not have any graphical elements for settings, then *getSettingsComponent* returns *null* and steps 2-3 are skipped. In case of repair algorithms an object of type *DecomposedModel*, which holds parts of the initial model and an event log for each of the parts, is returned.

4.4 Usage of decomposition and repair algorithms

A number of algorithms for conformance checking is really limited in ProM. There are only 2 prominent algorithms: conformance by replay and conformance using alignments, others are mainly variations of mentioned. Thus, there is no urgent need to provide really flexible solution. Both of these algorithms are used in the tool. The algorithm described in [3] is used as a main conformance algorithm in the tool, it is placed in *Conformance checking cell*. In order to allow convenient and user-friendly

usage of this algorithm, the corresponding plugin has been changed slightly. The plugin was partly separated from ProM in order to ensure its robustness. Moreover, parameters of the plugin include information on a model which is about to be used and the original parameter creation mechanism does not permit to create it silently, without showing GUI. Because of that reason, parameter classes were supplemented with "copy constructors" which take a new model and copy an existing parameter adhering it to the new model. Another algorithm is provided as an optional add-on and used in a *final evaluation cell*. The usage of this plugin required to slightly change classes related to user interface.

All discussed cells are parts of the abstraction called *repair chain*. A repair chain represents the very nature of the decomposed repair approach. Each chain implies algorithms which correspond to the cell types and then it makes plugin calls in the specified order ensuring the work of the tool. The goal of designing repair chains was to make a good level of abstractions from which algorithms (cells) are used, how they are used, in which order; and to execute every chain with different models without need to reconstruct the chain. In order to achieve these objectives, the idea of dependency injections is heavily exploited. Decomposition and repair plugins are supplied via constructor injection, whereas a model, an conformance checking algorithm and its parameters are provided as a method parameters. This discrepancy has rather ordinary explanation. Decomposition and repair algorithms represent something stable which can be reused over and over again with different models in a handy manner. In contrast, a model, a conformance checking algorithm and conformance checking parameters are volatile and tightly coupled.

Introducing a new data type which encapsulates cells tend to make the tool more flexible and easier to modify, maintain and extend because of the following reasons.

Using abstract data types and dependency injection during the development ensured that each particular chain may be implemented in a way which differs a lot from others. For instance, repair chains may use different triggers to decide when a repaired model is good enough, although the main reason for having separate repair chains is a fact that there are a few of possible strategies of how to choose a model part to be repaired. Some strategies are straightforward – just take a part with the worst fitness, whereas others may use sophisticated techniques, preprocessing and more intelligent choice. However, details, ins and outs of these strategies are out of scope of the paper due to their theoretical nature, the main point here is to establish that different repair chains are possible and that the tool has to provide capability of introducing new repair chains.

It allows users to create several chains which differ in algorithms used in cells and then run all of them at a time. The feature makes testing of several algorithms and their parameters against the same model a lot faster. In order to achieve it 2 plugins are available. One of them, *Iskra chain generator* is responsible for creating repair chains – one selects desired repair chains, algorithms and their parameters. In contrast with a main plugin which creates a chain and then immediately executes it, chain generator returns a list of configured chains to the ProM resource pool rather

than execute them. At the moment when all desired chains are built, one may supply them to *Iskra chain runner* plugin. This plugin takes an arbitrary number of repair chains, a model and a corresponding event log, after that the plugin configures settings of conformance checking and sequentially executes each chain. This functionality has already been implemented, although it needs some refinement and improvements.

In order not to have hard-coded chains and plugins around chains a mechanism of annotations and reflective calls was introduced, as used for decomposition and repair wrappers. It enriches the tool with the ability to load repair chains dynamically. Moreover, it lets other developers and researcher to develop new chains, incorporate them in the tool. A Java class which implements repair chain logic has to extend *RepairChain* interface and be annotated with *@Chain*.

4. Use case

As an example of a usage a simplified version of an iteration of a typical agile development process is considered. All activities of the developers are recorded in event log, thus allowing for keeping track of what the team does and analysis of the development process. Initial business process involves writing and running tests after writing code is completed. Then, a developers team informally decides to try test-driven development [18], thus creating tests before writing code. These changes are reflected in event logs. After a while a conformance checking algorithm is applied and it reveals that the actual process does not conform to what a company considers as an actual process. Hence, it is necessary to apply repair algorithm in order to learn what has changed and build a proper model of the process.

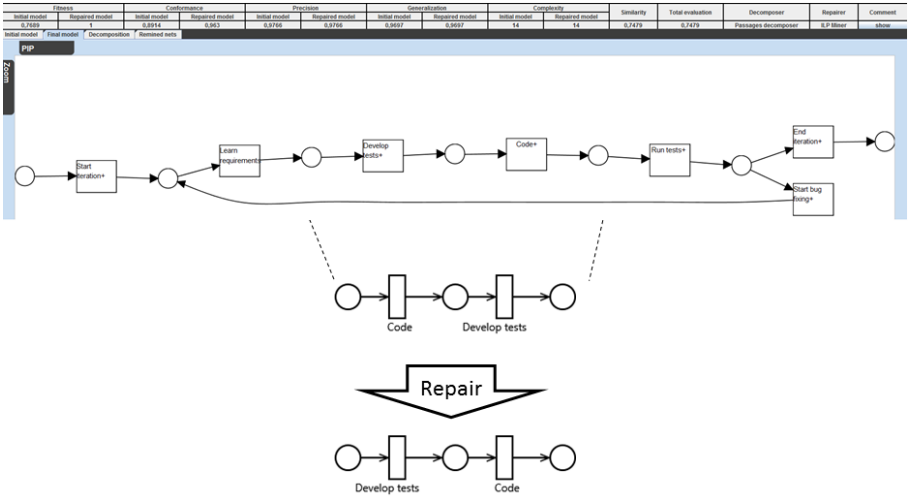

Fig. 3. Illustration of repair

In order to repair a model one needs to select appropriate plugin and supply an existing model and an event log. Plugin's graphical interface used for specifying settings is shown in figure 4. Then one selects desired algorithms of decomposition and repair. Moreover, one sets minimal fitness a repaired model should have. In the example, desired fitness is 0.98. The next step is to select an appropriate repair chain from the list of chains. Afterwards, one is asked to specify setting of each selected algorithm, and after that repair process is executed. Once it is finished, a screen with results is shown; it looks like in figure 3. This screenshot demonstrates the result of repair and final evaluation of the considered example of agile iteration and clarifies where the change took place and what exactly has changed. As a modeling language Petri nets are applied. It is clear from the screenshot that fitness increased from 0.7689 to 1, which means that the repair model perfectly fits the given event log and the goal of achieving fitness not smaller than 0.98 has been successfully accomplished. Furthermore, values of others metrics are shown on this screen.

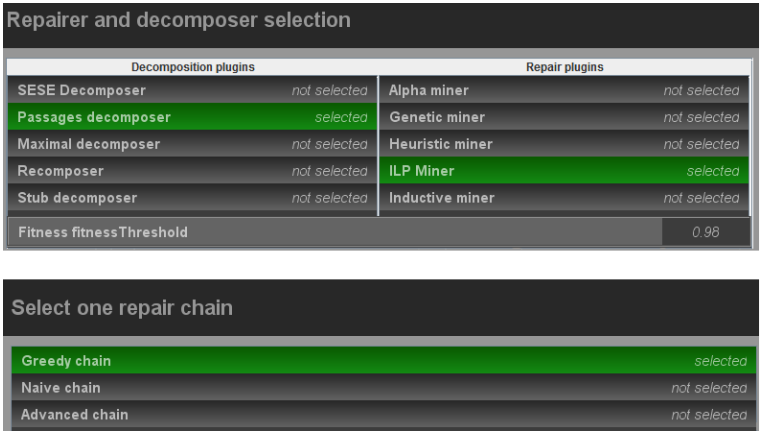

Fig. 4. Plugin settings

5. Related Work

The idea of providing a way to chain executions of several plugins or algorithms in a handy way, which is explored in this paper, is also similar to scientific workflow systems. Two of such systems capable of dealing with process mining are considered here.

First tool is *RapidProM* [19] which is a ProM extension for *RapidMiner* [20]. It allows users to build plugin chains in a visual way. Quite a number of ProM Plugins are available in this extension, however not all of them. It can easily be installed via RapidMiner Marketplace. The only question is its ability to be extended.

RapidProM does not support native ProM plugins and ProM mechanism for loading plugins, therefore plugins come only from the authors of RapidProM, which makes the objective of creation and execution of schemes, such as those discussed in the paper and possible in the presented tool, much harder.

Then comes *DPMine Workflow Language* [21] and *DPMine/P framework* which provide a new modeling language which natively supports notion of execution. Implementation of the ideas defined in the language is written in C++ with usage of Qt library. Process models can be constructed using convenient graphical user interface. Furthermore, the solution is intended to be easily extended by adding plugins. The advantage of using C++ is possibility to utilize resources in more effective and flexible way and provide better performance, which is of high importance in the era of Big Data, but the downside is that it cannot be integrated with ProM, so it is deprived of algorithms the ProM system offers.

6. Conclusion

In this paper, a tool for decomposed model repair is described. Decomposed model repair is used as a way of model enhancement. The tool is implemented as several plugins for the ProM Framework, which guarantees that the tool can be easily distributed and used by both researchers and developers within ProM community. The way the tool is written allows for fast improvement and enhancement of it.

While developing the tool advantages and disadvantages of existing tools were examined. The tool does not have some drawbacks typical for its counterparts. However, there is still room for improvements. In the future the tool will be fitted with more sophisticated mechanism of repair chains. Furthermore, a handy visualization of differences between initial and repaired models, some kind of recommender systems which suggests better repair options according to properties of a model and an event log will possibly be developed and incorporated.

Acknowledgement

This work is output of a research project implemented as a 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. 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. van der Aalst, A. Adriansyah, and B. F. van Dongen. Replaying history on process models for conformance checking and performance analysis. Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery, vol. 2, no. 2, pp. 182–192, 2012.
[4]. A. Rozinat and W. M. van der Aalst. Conformance checking of processes based on monitoring real behavior. Information Systems, vol. 33, no. 1, pp. 64–95, 2008.
[5]. D. Fahland and W. van der Aalst. Repairing process models to reflect reality. Business Process Management, ser. Lecture Notes in Computer Science, A. Barros, A. Gal, and E. Kindler, Eds. Springer Berlin Heidelberg, 2012, vol. 7481, pp. 229–245.
[6]. 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.
[7]. W. M. Van Der Aalst. A general divide and conquer approach for process mining. Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on. IEEE, 2013, pp. 1–10.
[8]. Prom framework. [Online]. Available: http://www.promtools.org/doku.php, accessed 2015-06-25.
[9]. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-oriented Software Architecture: A System of Patterns. New York, NY, USA: John Wiley & Sons, Inc., 1996.
[10]. W. van der Aalst, T. 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.
[11]. J. van der Werf, B. van Dongen, C. Hurkens, and A. Serebrenik. Process discovery using integer linear programming. Applications and Theory of Petri Nets, ser. Lecture Notes in Computer Science, K. van Hee and R. Valk, Eds. Springer Berlin Heidelberg, 2008, vol. 5062, pp. 368–387.
[12]. W. van der Aalst, A. de Medeiros, and A. Weijters. Genetic process mining. Applications and Theory of Petri Nets 2005, ser. Lecture Notes in Computer Science, G. Ciardo and P. Darondeau, Eds. Springer Berlin Heidelberg, 2005, vol. 3536, pp. 48–69.
[13]. A. Weijters, W. M. van Der Aalst, and A. A. De Medeiros. Process mining with the heuristics miner-algorithm. Technische Universiteit Eindhoven, Tech. Rep. WP, vol. 166, pp. 1–34, 2006.
[14]. S. Leemans, D. Fahland, and W. van der Aalst. Discovering blockstructured process models from incomplete event logs. Application and Theory of Petri Nets and Concurrency, ser. Lecture Notes in Computer Science, G. Ciardo and E. Kindler, Eds. Springer International Publishing, 2014, vol. 8489, pp. 91–110.
[15]. J. Munoz-Gama, J. Carmona, and W. M. van der Aalst. Single-entry single-exit decomposed conformance checking. Information Systems, vol. 46, pp. 102 – 122, 2014.
[16]. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
[17]. W. van der Aalst. Decomposing process mining problems using passages. Application and Theory of Petri Nets, ser. Lecture Notes in Computer Science, S. Haddad and L. Pomello, Eds. Springer Berlin Heidelberg, 2012, vol. 7347, pp. 72–91.
[18]. Beck. Test Driven Development: By Example. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
[19]. Rapidprom. [Online]. Available: http://www.rapidprom.org/, accessed 2015-06-25.
[20]. Rapidminer. [Online]. Available: https://rapidminer.com/, accessed 2015-06-25.

[21]. S. Shershakov. DPMine/C: C++ library and graphical frontend for DPMine workflow language. Proceedings of the Spring/Summer Young Researchers Colloquium on Software Engineering, vol. 8, 2014.

Iskra: Инструмент починки моделей процессов

И. Шугуров <shugurov94@gmail.com>,
А. Мицюк < amitsyuk@hse.ru>,
Лаборатория процессно-ориентированных информационных систем,
Национальный Исследовательский Университет «Высшая Школа Экономики», Россия, г. Москва, Кочновский пр., д. 3.

Аннотация. В данной работе представлена программа для проведения экспериментов в области process mining и тестирования алгоритмов починки моделей. Исследователи в области Process mining разрабатывают и применяют алгоритмы и подходы для извлечения и анализа бизнес процессов, которые основаны на анализе логов событий. Выделяют три основных области в рамках process mining: извлечение процессов, проверка соответствия моделей и логов событий и усовершенствование моделей. В данной статье рассматривается один из способов усовершенствования моделей, называемый починкой моделей процессов. Починка модели процесса необходима в случаях недостаточного соответствия существующей модели заданным логам событий реального процесса. Предполагается, что логи событий отражает правильное и актуальное поведение бизнес-процессов, в то время как модели процесса могут быть ошибочными. В статье рассматривается реализация модульного подхода для починки моделей. Предлагаемый подход предполагает реализацию программы, состоящей из нескольких независимых модулей, реализующих различные этапы процесса починки модели процесса. Подобная архитектура позволяет добиться более гибкой конфигурации починки, а также обеспечивает возможность проведения экспериментов по выбору алгоритмов, применяющихся в каком-либо модуле, в изоляции от других модулей. Несмотря на то, что основной целью статьи было описание особенностей реализации программы, теоретические основы модульной починки моделей процессов рассмотрены на уровне, достаточном для понимания подхода. Более того, рассмотрены сценарии использования программы и описаны способы её расширения дополнительными алгоритмами и функционалом. Приведен обзор существующих модульных решений, которые могут быть использованы для усовершенствования моделей процессов, обсуждены их достоинства и недостатки.

Keywords: Process model, Petri net, Model repair, Process mining.

DOI: 10.15514/ISPRAS-2015-27(3)-16

Для цитирования: Шугуров И., Мицюк А. Iskra: Инструмент починки моделей процессов. Труды ИСП РАН, том 27, вып. 3, 2015 г., стр. 237-254 (на английском языке). DOI: 10.15514/ISPRAS-2015-27(3)-16.

Список литературы

[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. 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. van der Aalst, A. Adriansyah, and B. F. van Dongen. Replaying history on process models for conformance checking and performance analysis. Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery, vol. 2, no. 2, pp. 182–192, 2012.

[4]. A. Rozinat and W. M. van der Aalst. Conformance checking of processes based on monitoring real behavior. Information Systems, vol. 33, no. 1, pp. 64–95, 2008.

[5]. D. Fahland and W. van der Aalst. Repairing process models to reflect reality. Business Process Management, ser. Lecture Notes in Computer Science, A. Barros, A. Gal, and E. Kindler, Eds. Springer Berlin Heidelberg, 2012, vol. 7481, pp. 229–245.

[6]. 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.

[7]. W. M. Van Der Aalst. A general divide and conquer approach for process mining. Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on. IEEE, 2013, pp. 1–10.

[8]. Prom framework. [Online]. Available: http://www.promtools.org/doku.php, accessed 2015-06-25.

[9]. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-oriented Software Architecture: A System of Patterns. New York, NY, USA: John Wiley & Sons, Inc., 1996.

[10]. W. van der Aalst, T. 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.

[11]. J. van der Werf, B. van Dongen, C. Hurkens, and A. Serebrenik. Process discovery using integer linear programming. Applications and Theory of Petri Nets, ser. Lecture Notes in Computer Science, K. van Hee and R. Valk, Eds. Springer Berlin Heidelberg, 2008, vol. 5062, pp. 368–387.

[12]. W. van der Aalst, A. de Medeiros, and A. Weijters. Genetic process mining. Applications and Theory of Petri Nets 2005, ser. Lecture Notes in Computer Science, G. Ciardo and P. Darondeau, Eds. Springer Berlin Heidelberg, 2005, vol. 3536, pp. 48–69.

[13]. A. Weijters, W. M. van Der Aalst, and A. A. De Medeiros. Process mining with the heuristics miner-algorithm. Technische Universiteit Eindhoven, Tech. Rep. WP, vol. 166, pp. 1–34, 2006.

[14]. S. Leemans, D. Fahland, and W. van der Aalst. Discovering blockstructured process models from incomplete event logs. Application and Theory of Petri Nets and Concurrency, ser. Lecture Notes in Computer Science, G. Ciardo and E. Kindler, Eds. Springer International Publishing, 2014, vol. 8489, pp. 91–110.

[15]. J. Munoz-Gama, J. Carmona, and W. M. van der Aalst. Single-entry single-exit decomposed conformance checking. Information Systems, vol. 46, pp. 102 – 122, 2014.

[16]. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
[17]. W. van der Aalst. Decomposing process mining problems using passages. Application and Theory of Petri Nets, ser. Lecture Notes in Computer Science, S. Haddad and L. Pomello, Eds. Springer Berlin Heidelberg, 2012, vol. 7347, pp. 72–91.
[18]. Beck. Test Driven Development: By Example. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
[19]. Rapidprom. [Online]. Available: http://www.rapidprom.org/, accessed 2015-06-25.
[20]. Rapidminer. [Online]. Available: https://rapidminer.com/, accessed 2015-06-25.
[21]. S. Shershakov. DPMine/C: C++ library and graphical frontend for DPMine workflow language. Proceedings of the Spring/Summer Young Researchers Colloquium on Software Engineering, vol. 8, 2014.