ISSN 0146-4116, Automatic Control and Computer Sciences, 2016, Vol. 50, No. 7, pp. 477–485. © Allerton Press, Inc., 2016.
Original Russian Text © S.A. Shershakov, 2014, published in Modelirovanie i Analiz Informatsionnykh Sistem, 2014, Vol. 21, No. 5, pp. 102–115.

DPMine Graphical Language for Automation of Experiments in Process Mining

S. A. Shershakov

National Research University Higher School of Economics, ul. Myasnitskaya 20, Moscow, 101000 Russia
e-mail: sshershakov@hse.ru

Received October 1, 2014

Abstract—Process mining is a new direction in the field of modeling and analysis of processes, where an important role is played by the use of information from event logs that describe the history of the system behavior. Methods and approaches used in process mining are often based on various heuristics, and experiments with large event logs are crucial for substantiating and comparing developed methods and algorithms. These experiments are very time consuming, so automation of experiments is an important task in the field of process mining. This paper presents the DPMine language developed specifically to describe and carry out process mining experiments. The basic concepts of the DPMine language as well as principles and mechanisms of its extension are described. Ways of integration of the DPMine language as dynamically loaded components into the VTMine modeling tool are considered. A sample experiment of building a fuzzy model of a process from a data log stored in a normalized database is given.

Keywords: process mining, modeling language, automation, experiments, fuzzy model

DOI: 10.3103/S014641161607018X

INTRODUCTION

This paper is about the simulation of experiments in the field of process-aware information systems that are being intensively developed due to changes in approaches to managing business and process systems.

Information system processes are becoming increasingly complex, and the amount of related information is steadily increasing. As a result, the number of studies related to Big Data processing is experiencing headlong growth [1]. We have recently seen the emergence of new specialties in the field of data and process research, such as data analyst, process researcher, process engineer [2], etc.

In English-language sources, the new area of process research is referred to as *process mining,* which consists of process retrieval (automated drawing) and analysis [3]. The suggested reference point for process studies in this area is *event logs*, which are automatically generated by most process-aware information systems. An *event log* is a track left by the information system during operation. Usually, this log consists of a set of *event* sequences. A single event sequence is called a trace, which is a record of process execution in one specific *case*. Each event in the trace marks the completion of a so called *activity.* An activity is a single task or phase of the considered process.

Process mining is closely related to data mining, machine learning, and process model simulation and analysis. The main goals and objectives of this research area are expounded in the Process Mining Manifesto [4]. They can be briefly summarized in the following three key problems: (1) process discovery; (2) conformance checking; and (3) model enhancement, which takes account of variable data.

Although process mining is a relatively new area, it is already actively used to model and analyze business processes [5] in management, software development [6, 7], process data management, and healthcare.

The process-mining theory is based on formal models and algorithms. However, the study of specific domain areas usually involves a large number of empirical conjectures and assumptions. This determines the domain- and application-oriented nature of these studies, which makes it necessary to carry out a lot of experiments to assess particular methods for applicability. These experiments can involve using not only real system logs, but also logs synthesized by models of processes represented, e.g., as Petri nets [8, 9].

Process models are described by various means, such as workflow notations and workflow engines/systems. A large number of currently available notations are used to describe processes, task flows (most of which are based on Petri nets), and systems, where these notations are implemented as tools for end users. Some common instances of these means are business process model and notation (BPMN) [10], business process execution language (BPEL) [11], and yet another workflow language (YAWL) [12].

To describe a process mining experiment, as well as other processes, a record of this experiment must be made using a certain notation. This record can be an experimental model or an experimental program. This model can be recorded in one of the foregoing languages; however, this approach has several weak points. First of all, they result from the fact that some of available modeling languages are intended for a specialized domain, e.g., business-process management or web-service orchestration. Other languages are more universal but do not always make it possible to describe the experimental model in a user-friendly concise manner.

It was proposed to automatize process mining experiments by using an approach based on the DPMine graphical extendable modeling language [13]. The requirements that imposed the development of the new language and related tools for its implementation included the existence of simple, transparent, flexible, and most importantly, extendable semantics. The DPMine language was initially developed for this particular purpose and was implemented as a set of plug-ins [14] for the ProM process mining toolkit [15]. ProM is one of the most functionally ample process mining tools.

Despite the main purpose of DPMine, experimental process mining and analysis are not its only applications. The language can also be used to automatize different software-programmable processes by means of an extendable block system. Unlike the YAWL with a purpose similar at first sight [12], the DPMine language does not have an integral part that consists of a declared general set of blocks/tasks for support of stream-controlling constructs, e.g., XOR/AND split/join. Instead, DPMine supports a flexible system of expanding block types that endow the language with any necessary semantics, including standard control constructs.

This paper considers the key features of the DPMine language implemented as the DPMine/C library in the C++ language [16]. This library is integrated with the VTMine universal expandable modeling system [17]. Correspondingly, VTMine is the target environment of this language. The key solutions for actualizing the expandable structure of both tools that work in close relation with one another are considered. A sample experiment in DPMine is considered, which is conducted to automatically draw the process model as a fuzzy map that proceeds from an event log represented as a normalized database.

The rest of the paper is structured as described below. Section 1 reviews the main concepts, components, and principles for the implementation of DPMine. Section 2 discusses the approaches to integrating the DPMine/C library with the VTMine modeling tool. Section 3 describes a case for creating a model process mining experiment. Finally, Section 4 tallies up the works completed and formulates tasks for further investigation.

1. BASIC LANGUAGE COMPONENTS

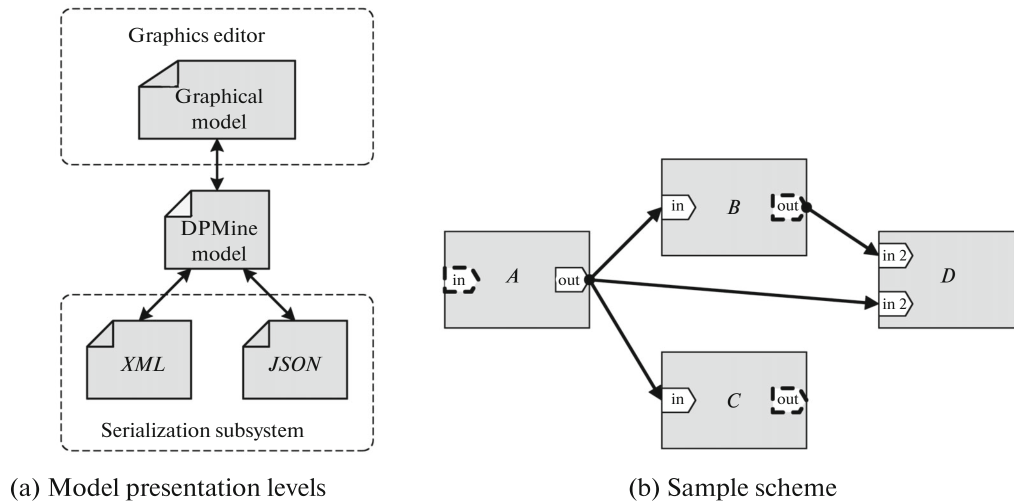
DPMine is a graphical language for description of processes and experiments that consist of individual components (blocks). The system of types of these blocks is extendable, which makes it possible to use this language to describe processes in different problem spaces [13, 14]. This section considers the main conceptual elements of the DPMine language.

1.1. Models, Schemes, Blocks, Ports, Connectors

The model is the basic working language tool that describes some processes, such as task workflows. In the final outcome of this process all or certain tasks included in the model are fulfilled. The model is considered at different levels of abstraction (Fig. 1). At the bottom level, we have the object model that represents the structure of complementary objects in the random access memory. The graphical model is visually represented as an editable graphical scheme. The persistent model is used in the serialization of the model, e.g., in the form of files.

The model includes the main scheme and attributes, such as name, version, etc. The model is actualized by the basic tool, which is an application with DPMine/C library as its client. The tasks described by the model are defined by the purpose of the basic tool. Instances of these tasks are creation, transformation, visualization, and serialization of formal models.

The language semantics is actualized via the concept of *blocks, ports, connectors,* and schemes*s*. Any expansion of the language is only via these four components.



(a) Model presentation levels

(b) Sample scheme

Fig. 1. Model and scheme.

A *block* is the key language element and considered to be an elementary operation. There are different types of blocks that can therefore be used to fulfill tasks of the basic tool, e.g., by addressing a particular method or a plug-in, represent complex system as a single hierarchical block, actualize structures of controlling the execution workflow. In addition, blocks can be used as substitution operators to transfer a certain scheme as a parameter to another scheme, etc. Blocks are integrated by executing tasks in a hierarchy of types.

A *port* is a connection object that belongs to a block and possesses the characteristics of direction (input, output, proxy ports) and data type. Ports are used to transport to and from the block objects (resources) of a given type. Ports are divided by block type into custom ports and built-in ports.

A *connector* is a directed connection object that connects two blocks via their ports; it connects the output port of one block via its origin to the access port of the other block via its end. There can be several connectors coming out of one output port but only one connector going into one access port (Fig. 1b).

A *scheme* is a set of related blocks connected by connectors. The scheme is the main way of actualizing abstraction and isolating the subprocess and hierarchy. The scheme interface is a random subset of ports Ifp of the set of all ports. This set is formed by the blocks included in this scheme. In Fig. 1b, the scheme interface is represented by ports `Ifp = {A.in, B.out, C.out}` (highlighted with a dashed outline).

At the object model level, the scheme is packed in a container represented by a special scheme-type block. This block inherits the characteristics of the basic type, i.e., *bodyness block,* which is the parent block for other blocks (body) encapsulated within it. In addition to the scheme, the bodyness block is the parent block for types, such as cycles.

The bodyness block can be considered at two levels, i.e., internally and externally. Internally, this block is an isolated scheme with an interface linked via connectors to the proxy ports of the block that contains this scheme. The ports are used in data exchange with external blocks. Externally, the bodyness block is a regular block with input and output ports (they also function as internal-level proxy ports) that exchanges data with sibling blocks at the same hierarchy level. This approach makes it possible to consider the scheme as the single block and to exclude the contents from consideration.

1.2. Execution of Model, Scheme, Block

The DPMine model is executable. This is done by executing the main scheme, which is determined by the composition of the blocks in the scheme and the structure of their connection. When the model is executed, the blocks included in the model create new resources and transform already available resources, carry out preparatory operations for visualization and saving, etc.

The model is executed by a special executive agent that connects the DPMine modeling system with the basic tool.

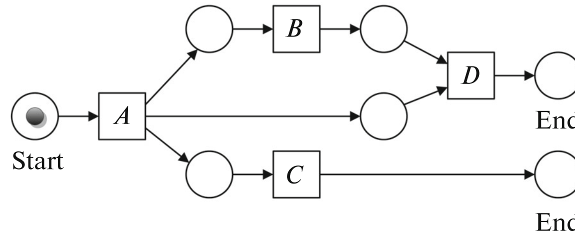


Fig. 2. Petri net model for implementing the scheme from Fig. 1b.

The main model scheme is executed by executing its container block. *Block execution* is an operation executed in software associated with the type that determines each block of the DPMine language. A case of this operation is the algorithm that receives resources from the access block ports as parameters, makes new resources based on these parameters, and places the new resources at the output ports of this block (e.g., when a Petri net is transformed into a spanning graph of this network).

The block can be executed under the condition that all of its external dependences are satisfied. The dependences of block *B* are considered satisfied given the following:

(1) *B* has no access ports.

(2) The block does have access ports, each of which meets the following conditions:

(2a) It does not have the *mandatory connected* flag, in which case the port cannot be linked by the connector to the other (output) port of another block.

(2b) It is connected to the other (output) port of another block with the *executed* status. As required by this status, the output ports of this block must contain data for implementing the output resource interfaces declared in the block type.

The status of the block with the satisfied external dependences is *executable*. In this case, the executive agent is the procedure for executing the block, and the status of the latter is changed to *in execution*. If the block is successfully executed, its status is changed to *executed*.

The bodyness block has an iterative execution algorithm. In each iteration the executive agent tries to execute a subset of executable blocks of the set of all the blocks included in the bodyness block scheme. During operation, the algorithm determines and uses several special status flags for the complete scheme. For instance, the *incomplete* flag shows that there are some blocks that are still unexecuted after a regular iteration. The *hasExecution* flag shows that at least one block was executed during a regular iteration. Finally, the *hasPending* flag shows that some blocks are still in execution by the end of a regular iteration. These blocks are called *pending*.

In the initial operation phase of the algorithm all the three flags are reset. The algorithm tries to execute each block of the scheme. If the *observable* block has already been executed, the algorithm skips it and goes on to the next block. If the observable block is pending, a *hasPending* flag is placed and the algorithm goes on to the next block. If the observable block must be executed in the current iteration, the system checks whether the input dependences of the block are satisfied. If yes, the block is passed on to the executive agent for dispatching. If the iteration is finished but there is still at least one block with a status other than *executed,* i.e., its execution has not started or the block is pending, the block is given an *incomplete* flag. In the latter case, the algorithm carries out yet another operation if at least one block was put on execution in the past iteration.

The scheme execution procedure can be presented as a Petri net model. For instance, the equivalent Petri net for the scheme without selection blocks (Fig. 1b) can be drawn as shown in Fig. 2.

2. INTEGRATION OF THE DPMINE LANGUAGE WITH THE VTMINE MODELING SYSTEM

VTMine is a graphical modeling tool with functionalities extended by dynamically loaded components called *plug-ins* [17].

The VTMine core determines the minimal necessary plug-in loading algorithms and declares interfaces (including some components of the user interface). All the other capabilities are completely deter-

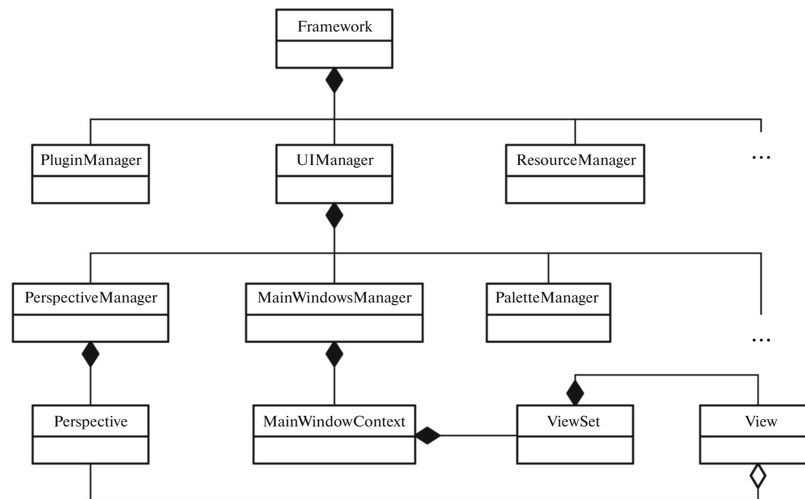


Fig. 3. Basic components of the VTMine core.

mined by the set of plug-ins. The main models for consideration are formalisms used most often in process mining, including graph structures, transition systems, finite state machines, Petri nets, etc. However, the system core does not have any special support for particular classes of models. That is why VTMine can be used for modeling in various domains.

VTMine is written in C++: on the one hand, this makes it possible to exercise flexible control over memory distribution; on the other hand, it becomes possible to develop fast and efficient algorithms, which is especially important in big data processing. The application is based on the Qt cross-platform library: the actively used subsystems of Qt include GUI used to build the user interface, the object-oriented subsystem of plug-in loading that encapsulates interaction with initial formats of dynamic link libraries of each OS individually, and the Graphics View framework high-performance subsystem of flowchart drawing.

2.1. Application Core Structure

The application core is a hierarchical system of basic components called *managers* (Fig. 3). This structure makes it possible to modify individual components of both the core and plug-ins without losing in compatibility among different versions in the phase of load-time linking. Any component in the system can be replaced with another component that actualizes the same interface.

The main application component is the root object Framework: it forms the *framework* of the application. This object contains such top-level components and managers as logger, Plug-inManager, UIManager, ResourceManager, etc. The framework is available in most modules and transferred as a parameter to the access point of each plug-in, which makes it possible for such plug-ins to modify top-level components.

The *plug-in control system* controls the life cycle of VTMine plug-ins. This cycle includes identifying plug-in file containers, loading plug-ins with regard to their interconnection, registering plug-ins in the system, using plug-ins to modify other plug-ins, and swapping plug-ins.

VTMine takes into account the interconnection among plug-ins. This means that, if plug-in P_A depends on P_B (both plug-ins have unique string identifiers), then P_A can only be loaded if P_B is also loaded. Consequently, if P_B cannot be loaded for any reason whatsoever, P_A will not be loaded either. A particular case is circularly dependent plug-ins, neither of which is loadable.

2.2. DPMine Plug-ins

The DPMine modeling system is connected to the VTMine application as a set of plug-ins with DPMineBase as the basic plug-in. The structure of the basic components of the plug-in is given in Fig. 4.

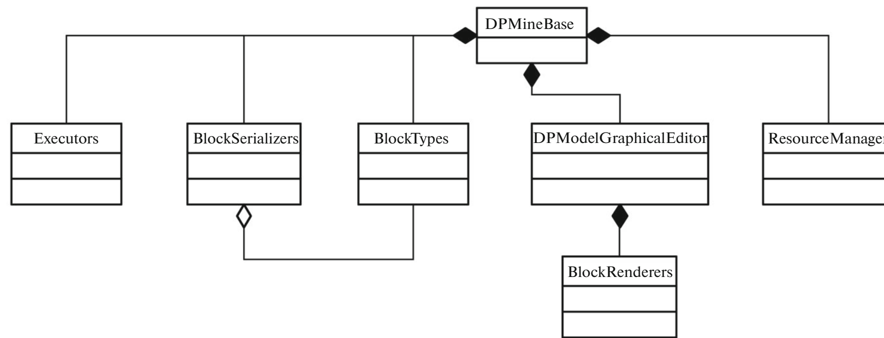


Fig. 4. Basic components of the DPMineBase plug-in.

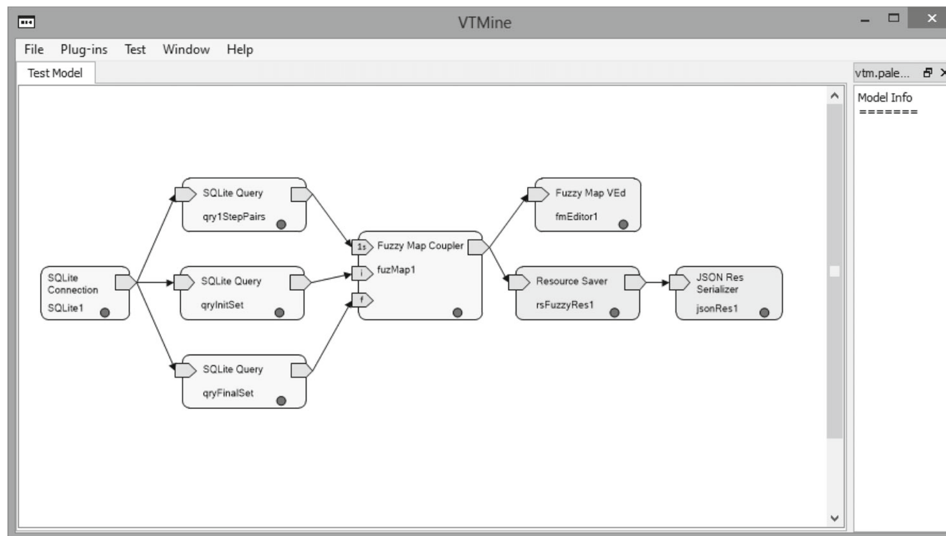


Fig. 5. Display form of VTMine application with a view of the DPMine model.

The *resource manager* registers types of resources of the DPMine language, such as the DPMine model, and registers new doers for certain types of resources and actualizes the functionalities of these doers.

The *list of block types* stores descriptions, descriptors, and other necessary information on block types that can be used to create/execute the DPMine model. The list is dynamically expanding and can be supplemented with new types using other plug-ins. The main plug-in (DPMine) registers only basic block types such as the scheme, execution flow control blocks, and blocks for implementing constants and equations. Blocks specific to a particular domain, e.g., process mining, are registered using other plug-ins.

Serializers are dynamically expanding components that load or save blocks from a certain serialization format, e.g., XML or JSON. The interface provided by a serialization block makes it possible to register both new serialization formats and relations among formats and individual block types using block serializers.

The *graphics editor* is a visual component responsible for the communication at the UI level between the DMPine models and the VTMine application. The editor makes it possible to create, modify, and execute the DPMine models presented in a graphical form (Fig. 5).

In the VTMine application, the editor is presented as a view created and added to the list of views of the active window. For instance, this operation can be carried out by applying a doer called the *graphics editor of DPMine models* to a resource like the *DPMine model*. This model is treated like other types of models, such as graphs and Petri nets.

The communication between the user and the graphical model is managed by contextual commands for each block. For instance, special-type blocks used to visualize the model's resources contain a doer of some command. When fetched, this command creates a new view (in VTMine terms) that contains a representation of the resource (usually graphical).

The visual display of any of the model blocks is determined by several factors such as block type, current block settings (for blocks with adjustable parameters), current block status, and selected rendering scheme.

3. DISCOVERY OF A FUZZY PROCESS MAP FROM AN EVENT LOG STORED IN A DB

In [18], an approach to retrieving a fuzzy map [19] from a data log presented as a relation database (RDB) is considered. This approach is based on a sequence of SQL queries that form necessary data sets to visualize this map.

The considered approach can be implemented as a DPMine model with the following blocks by type/name (Fig. 5):

(1) SQLite Connection/SQLite1 is the data source in the SQLite RDB;

(2) SQLite Query/qry1StepPairs, qryInitSet, qryFinalSet are tabular data sets formed by answering the queries for: (a) the relations among all activities in one step, (b) the set of initial vertices, and (c) the set of terminal vertices;

(3) Fuzzy Map Coupler/fuzMap1 unites the three above described data sources in a single (object) fuzzy model;

(4) Fuzzy Map VEd/fmEditor1 is the activation point of the graphic editor of fuzzy models, that connects the object model with the graphical model;

(5) Resource saver/rsFuzzyRes1 is the component for saving time resources in the global resource manager;

(6) JSON Res Serializer/jsonRes1 serializes (saves) the resource (fuzzy model) as a JSON file.

The scheme in Fig. 5 makes it possible to use several different block execution sequences, taking account of the requirements imposed by the semantics of executing DPMine language models.

The first block for execution is SQLite Connection/SQLite1, which gives access to the SQLite database [20]. The information contained in this block, such as file name, opening parameters, and other parameters described in the SQLite application program interface (API), is required to connect to this DB. If all parameters of SQLite Connection/SQLite1 make it possible to connect to the DB, the block in execution carries out this connection and allows one to use the DB by the blocks involved in connection with this.

The SQLite Query blocks are used to form SQL queries to the DB and present the results of these queries as a table with data that correspond to the IDataSet interface. The model in Fig. 5 makes use of three such blocks, namely, qry1StepPairs, qryInitSet, qryFinalSet. They make a sampling in the source database (that represents the log) to form three different datasets required to build the fuzzy model graph. The three datasets are a sampling of one-step transitions between separate activities included in the resulting set of data and the samplings of initial and terminal activities.

The Fuzzy Map Coupler block binds the three datasets into a single object model. The access port of this block must have the 1*s* (one-step transitions) prepared set of data so that the block can be successfully executed. The other two access ports *i* and *r* correspond to the initial and terminal positions, are optional, and can be ignored.

Fuzzy models are presented as a graph with activities as vertices and relations among these activities as arcs. Thus, the execution of this block forms an output resource in the form of graph that is determined by the adjacency list formed from the set of one-step transitions. The graph is supplemented with links from the artificial vertex *beginning* at vertices from the set of initial vertices (second data set) and with links from the set of terminal vertices to one more artificial vertex *end*.

The Fuzzy Map VEd block is used to visualize the derived object model. This block is based on the component editor of graph models that is modified for fuzzy models.

The execution of the Fuzzy Map Ved/fmEditor1 block results in checking the availability of the graph-editing component loaded as a separate plug-in in the VTMine application. However, this component is not mandatory for satisfying the dependences of the plug-ins that load the VTMine blocks considered in this section.

If this component is available, the necessary initialization is carried out and the block is given the *executed* status. This means that the block becomes available for interactive communication with the user, who can find and execute the *Display Editor* command in the context-sensitive menu. As a result, a new view with an embedded graphics editor will be created that will be added to the other views of the active window of the VTMine application.

The input of the Resource Saver/rsFuzzyRes1 block is connected right to the output of the Fuzzy Map Coupler/fuzMap1, exactly like the Fuzzy Map Ved/fmEditor1 block. This means that both consumer blocks of the object (fuzzy) model make use of the same resource copy (model).

By default, the resources produced by the blocks of the DPMine model are in the state of composition with the blocks that generate these resources and control their life span. Thus, except for some special blocks, these temporary resources are deleted during the repeated execution or deletion of the model itself. The Saver/rsFuzzyRes1block is used to save the copy of this resource of the model using the global resource manager for further use.

The output port of Resource Saver/rsFuzzyRes1 is connected to the access port of JSON Res Serializer/jsonRes1, which is the last block considered in this scheme. When the block is executed, an object that actualizes the IResource interface is transferred to the access port of JSON Res Serializer/jsonRes1 and saved as a serialized JSON file.

The JSON Res Serializer/jsonRes1 serialization component is extended by the types of resources it can serialize. This extension is similar to the extension of the serialization component of the DPMine model blocks.

As a result of executing the model, its main scheme (Fig. 5), and all of the model's blocks, the database with the event log is opened and three queries with the extraction of necessary datasets are formed. These sets are aggregated in the object Fuzzy model (resource of VTMIne), the graphical component for visualizing this model is provided, the copy of this model is saved as a permanent resource, and the model is recorded as a serialized JSON file.

4. CONCLUSIONS

This paper describes the DPMine language concept and its implementation as the DPMine/C library. The library is integrated with the VTMine modeling system by means of plug-ins.

The key task to be fulfilled is the development of new types of blocks for supporting a larger number of process mining algorithms. The VTMine tools should be improved for creating and analyzing DPMine models and traditional models used in process mining. It is necessary to carry out the work to support vectoring complex algorithm computations. Finally, there is the distinct task of integrating VTMine with current process mining solutions, such as ProM.

ACKNOWLEDGMENTS

This work is supported by the Basic Research Program at the National Research University Higher School of Economics (2014).

REFERENCES

1. Manyika, J., Chui, M., Brown, B., et al., *Big Data: The Next Frontier for Innovation, Competition, and Productivity,* 2011.
2. Accorsi, R., Damiani, E., and van der Aalst, W., Unleashing operational process mining (Dagstuhl Seminar 13481), *Dagstuhl Rep.,* 2014, vol. 3, no. 11, pp. 154–192.
3. van der Aalst, W.M.P., *Process Mining—Discovery, Conformance and Enhancement of Business Processes,* Springer, 2011, pp. I–XVI; 1–352.
4. IEEE Task Force on Process Mining: Process Mining Manifesto, *Lect. Notes Bus. Inf. Process.,* 2011, vol. 99, pp. 169–194.
5. Mitsyuk, A., Kalenkova, A., Shershakov, S., and van der Aalst, W., *Using process mining for the analysis of an e-trade system: A case study*, *Bus. Inf.,* 2014, vol. 3.
6. Rubin, V., Lomazova, I., and van der Aalst, W.M., *Agile development with software process mining*, *ICSSP,* 2014. Nanjing.
7. Rubin, V., Mitsyuk, A.A., Lomazova, I.A., and van der Aalst, W.M.P., Process mining can be applied to software too!, *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement,* New York: ACM, 2014.

8. Mitsyuk, A.A. and Shugurov, I.S., On process model synthesis based on event logs with noise, *Model. Anal. Inf. Sist.,* 2014, vol. 21, no. 4, pp. 181–198.
9. Shugurov, I. and Mitsyuk, A.A., Generation of a set of event logs with noise, *Proceedings of the 8th Spring/Summer Young Researchers Colloquium on Software Engineering, SYRCoSE 2014*, ISP RAS, 2014.
10. Object Management Group (OMG), *Business Process Model and Notation (BPMN), Version 2.0,* 2011.
11. Alves, A., Arkin, A., Askary, S., et al., *Web Services Business Process Execution Language Version 2.0 (OASIS Standard),* WS-BPEL TC OASIS, 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.
12. Van der Aalst, W.M.P. and ter Hofstede, A.H.M., YAWL: Yet another workflow language, *Inf. Syst.,* 2005, vol. 30, no. 4, pp. 245–275.
13. Shershakov, S., DPMine: Modeling and process mining tool, *Proceedings of the 7th Spring/Summer Young Researchers Colloquium on Software Engineering. SYRCoSE 2013,* ISP RAS, 2013.
14. Shershakov, S., DPMine/P: Modeling and process mining language and ProM plug-ins, *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, New York, 2013.
15. Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., and van der Aalst, W.M.P., ProM 6: The Process Mining Toolkit, *CEUR Workshop Proc.,* 2010, vol. 615, pp. 34–39.
16. Shershakov, S., DPMine/C: C++ library and graphical frontend for DPMine workflow language, *Proc. 8th Spring/Summer Young Researchers' Colloquium on Software Engineering, SYRCoSE 2014,* ISP RAS, 2014, pp. 96–101.
17. Kim, P., Bulanov, O., and Shershakov, S., Component-based VTMine/C framework: Not only modelling, *Proceedings of the 8th Spring/Summer Young Researchers Colloquium on Software Engineering, SYRCoSE 2014*, ISP RAS, 2014, pp. 102–107.
18. Shershakov, S.A., VTMine framework as applied to process mining modeling, *Int. J. Comput. Commun. Eng.* (in press).
19. Günther, C.W. and van der Aalst, W.M.P., Fuzzy mining: Adaptive process simplification based on multi-perspective metrics, *Proceedings of the 5th International Conference on Business Process Management, BPM'07,* Berlin, Heidelberg, 2007, pp. 328–343.
20. http://www.sqlite.org/.

Translated by S. Kuznetsov