DPMine/C: C++ Library and Graphical Frontend for DPMine Workflow Language

Sergey Shershakov
International Laboratory
of Process-Aware Information Systems (PAIS Lab)
National Research University Higher School of Economics
Moscow 101000, Russia
Email: sshershakov@hse.ru

***Abstract*—DPMine generic purpose workflow language is rooted in DPMine/P scientific workflow language and a set of plug-ins for ProM which originally were developed for convenient piping of different plug-ins within ProM framework. DPMine/C is a new version of DPMine workflow language and a C++ library. The main language concept was complemented by comprehensive analysis of DPMine/C model execution semantics. This paper also discusses approaches to the *block types extension* concept relying on development of new block type classes and customization of the *model storage subsystem*. Finally, we show an approach for implementation of a GUI frontend.**

***Keywords:* Workflow, Modelling Language, C++ Library, Extensible Tool, Process Mining, Processes, PAIS**

I. Introduction

Today there exists a wide variety of workflow notations. Some of them have a formal basis (such as Petri nets, finite state automata), others have vendor specific notations. Among them one can distinguish some notations that pretend to be industry standards. BPMN [1] and BPEL [2] are, perhaps, just the most known examples of such standards [3].

At the same time, being a standard does not mean being appropriate for description of all kinds of workflow models. As an example, we refer to the papers [4], [5] where a problem of piping components (plug-ins) of ProM heterogenous system arises. We had to obtain an ability to create a *scientific workflow model* that includes individual invocations of particular processing algorithms (implemented, for example, with the help of ProM plug-ins), cycles support, choice constructs and other controls of the execution thread. As a result, DPMine/P language with a simple, transparent, flexible and, most importantly, extensible semantics has been developed and implemented as a set of ProM plug-ins [5].

Typically, workflow management systems are used for maintaining various kinds of *business* processes. DPMine is not another Business Process Management (BPM) system. We rather call DPMine as a "technical workflow language".

Unlike most workflow languages DPMine has much more imperative rather than declarative nature. A model in DPMine language is similar to a program in some way. As well as a program, a model can be *executed*, thus we pay much attention to its execution semantics. We explicitly state that it is "execution", not just a "simulation". In the case of a well-formed algorithm the model execution "outcomes" are determined only by the model incoming *resources* and by the model structure (depending, of course, on the nature of the blocks contained in the model).

Along with the differences DPMine has also a lot of similarities with existing workflow languages. Thus, DPMine uses *ports* notations, just as BPEL (Web Services Business Process Execution Language, WS-BPEL) [6] does. But in DPMine the *port* is one of the main language building elements widely used for setting relations between *blocks* — another important building element.

There is a specific language family of Petri net based workflow languages [7]. One of the most known is YAWL — a workflow language extended with additional features to facilitate the modelling of complex workflows [8]. Unlike YAWL, DPMine does not introduce a generic set of tasks[1] that support control-flow tasks (such as AND/XOR split/join and so on) as a part of the language core. We proposed instead some control-flow blocks [5]. We considered them only as an example of custom block type implementation. In this paper we are primarily considering blocks as abstract objects.

Basically, the requirements for DPMine language have been indicated through the requirements imposed on a scientific workflow language for ProM tools. Moreover, one can say that DPMine/C has been emerging as a way of generalizing the solution of the piping task. One could consider incorporating one of the existing BPEL engines into ProM, but this approach is fraught with compatibility problems and leads to the use of BPEL in an area not much related to it. A similar situation happens with BPMN, but in this case the language application would be even more harder.

Workflow notations can be considered from different perspectives [9]. Here, we are mainly focusing on the *control-flow perspective*.

The rest of this paper is organized as follows. Section II describes the modular concept and the basic components of DPMine language. Section III discusses DPMine model execution semantics and approaches to resources transferring. Certain aspects of the extendable storage subsytem are presented in Section IV. Section V introduces an idea of implementation of a GUI frontend for DPMine language using

[1] They could be treated as so-called *task-blocks* in terms of DPMine.

This work is licensed under the Creative Commons Attribution License.

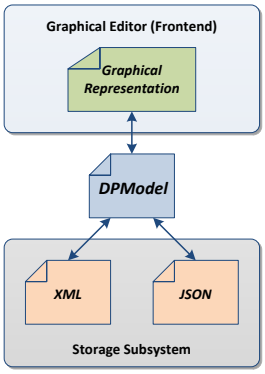


Figure 1. Levels of language representation

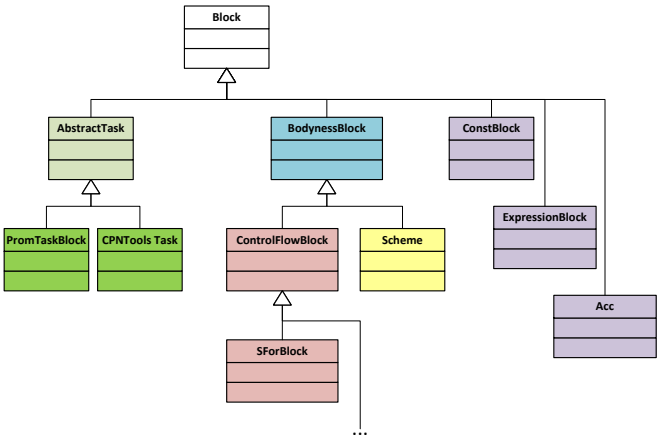


Figure 2. Blocks hierarchy scheme

Qt library. Finally, Section VI concludes with analysis of the work done and a look at the future.

II. DPMine Language Basic Elements

The main work concept for DPMine/C toolset is the *model*, which represents some *workflow model under experiment*. In a C++-based types system a *model* is represented as an object of `DPModel` base class containing all necessary information about the model such as model name, model author, and so on. The most important object contained in the model is the so-called *main scheme* to be executed during the model execution procedure (see sec. III). Developing a DPMine/C library client one can extend the `DPModel` defenition by adding project-relevant features.

The rest of this section is devoted to the model concept, its main components and their interaction.

A. Model Definition

A model can be considered from different points of view (Fig. 1). At the lower level there is a C++-based *object model* of the workflow model. At the medium level (it can also be referred to as a *storage level*) there is an XML-based model markup language or something similar such as JSON-based text format. Finally, at the upper level a model can be represented using a graphical notation, which allows defining the process model as a set of building blocks.

An object model can be serialized to an underlying XML- or JSON-based text file or deserialized from it (see. sec. IV). One can extend a serialization mechanism to utilize any other storage formats.

The graphical model is transformed into an object model and vice versa, and it can be used both for user-friendly model design and representing the model execution dynamic process.

B. Schemes, Blocks, Ports, and Connections Concept

Implementation of the basic language semantics is done through the concept of schemes, blocks, ports and connectors. Expansion of the language functionality should be based on this very concept. The main idea is that no special extensions for maintaining any kind of custom blocks are needed.

Let us examine these elements in more detail.

1) Block: It is a basic language building element considered as a solitary operation in an external representation but can be complex in internal one. Blocks perform a specific task and can be considered as statements in programming languages. Blocks can have different functionality such as performing a single task of a base platform (*task blocks*), representing complex schemes into single blocks (*scheme blocks*), implementing control workflow (*control flow* blocks) and so on.

The blocks are arranged by types into a hierarchy that is a reflection of corresponding C++ block types hierarchy, whose scheme is presented on Fig. 2.

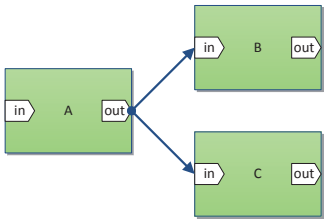
2) Port: It is an object belonging to a certain block and used for connection and data objects transfer to other ports. Depending on data flow direction, one can distinguish three types of ports: *input*, *output*, and so-called *proxy* (input-output and output-input). Ports transfer *resources* of a specific data type from one block to another. Depending on block type, they can be either custom or built-in.

3) Connector: It is an object connecting two blocks through their ports. Connectors have a *link direction*: a connector (with its beginning) always connects an output port of a block with an input port of another one (with its end). One output port can be linked to several connectors, whereas one input port can have only one connector linked (Fig. 3a).

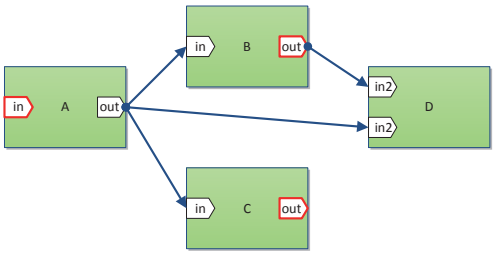
Starting from this implementation of DPMine library it was decided that it was not necessary to introduce a special data type representing connector at the programming level. Instead, we decided to use internal links between corresponding output and input ports. Such links can be treated as connectors at higher levels.

4) Scheme: It represents a number of interacting blocks connected with each other by connectors. The schemes are the main mechanism of implementing abstraction, isolation and hierarchy of sub-processes. On Fig. 3b there is depicted a connected scheme consisting of four blocks `(A, B, C, D)` and four connectors `(AB, AC, AD, BD)`.

One needs to implement a special container to represent a scheme at the object model level. Such a container is a



(a) Ports and connectors



(b) Scheme example

Figure 3. Various blocks, ports, connectors, and schemes elements

special type of block that is called *Scheme-block* (Fig. 2). The *Scheme-block* is a direct (and most obvious) descendant of a more general block type, namely *Bodyness block*. A *Bodyness block* is a special kind of blocks whose distinguishing feature is the ability to aggregate their so-called child-blocks inside themselves. In other words, the *Bodyness block* represented by `BodynessBlock` abstract class is the ancestor for all the block types that can own child blocks.

Any descendant of a *Bodyness block* (including *Scheme-blocks*) can be considered both at the external and the internal level. At the external level a *Bodyness block* is nothing more than just a regular block, which can have input and output ports that can be connected to some ports of some other blocks (at the same level). At the internal level the same block can be treated as an isolated scheme (maybe having some special behavior based on individual characteristics of the specific descendant[2]).

This scheme has the only way to communicate with external blocks at a higher level: by using its own port "everted" and represented at the internal level in the opposite direction. Thus, the ports viewed at the external level as input flip to be output at the internal level, and vice versa. This is why these ports are named "proxy".

III. Model Execution

One of the main goal for constructing a DPMine model is its subsequent execution. Outcome results of the executed model is the sum of results of its individual executed blocks. They are based on the subject domain of the task blocks contained in the model.

One can consider an example of workflow model containing some tasks that perform phased processing of a set of source tex-files with a view to obtaining a PDF-document. This produced PDF is such an outcome result.

Model execution consists in executing the *main scheme* of the model (upper level scheme) and producing an execution report (about errors, etc.). Model execution is done by a special agent — *Executor*, whose implementation is closely related to the client application design (see sec. III-C).

A. Block Execution

When considering the execution of a scheme one needs to mention the execution concept for an individual block. *Block execution* is an operation done by the underlying block's type class method `execute()` based on the block's individual state. In the example above there could be a special block type performing invocation of some LaTeX tool like `pdftolatex` as its execution procedure.

From the technical point of view `execute()` method is virtual, which means that it must be implemented individually for each type of blocks. It is also possible to modify the behavior of any previously defined block type by reimplementing this virtual method.

In order for a given block to be able to be executed it is necessary that all the external dependencies of the block be satisfied.

For a given block `B` its dependencies are considered satisfied if:

1) the block does not have input ports, or
2) the block has input ports and for each port the following conditions are met:
   a) there is no "must be connected" flag for the port set, this way the port can be not connected by a connector to another (output) port of another block;
   b) the input port is connected by a connector to another (output) port of another block and this output port is ready to give requested resource to the input port; in most cases, the latter means the status of parent block of the output port is "executed".

The block with satisfied dependencies is referred to as "executable" block.

B. Scheme Execution

As it was mentioned above every scheme is represented by its *Scheme-block*. So, speaking about the execution of a scheme one needs to consider the execution of its block.

Since the *Scheme-block* is a descendant of the *Bodyness block*, it does not define its own execution algorithm but utilizes an algorithm given by *Bodyness block* class[3].

In fact, this algorithm is the heart of DPMine execution semantics, so it has to be discussed in greater detail.

The mentioned algorithm is iterative. Some subset of the full set of scheme body blocks is tried to be executed in each iteration. During its execution the algorithm defines some

[2]For example, consider *For-loop block*, see [5].

[3]This also holds for many other *Bodyness block* descendants like *For-loop block*, etc.

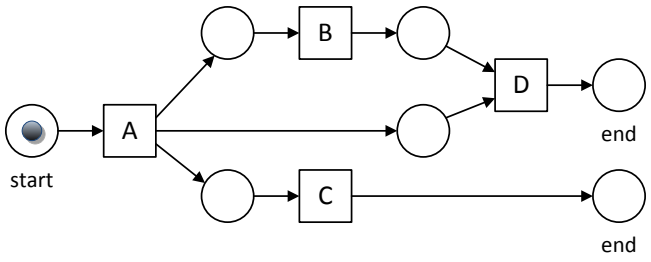


Figure 4. Equivalent (system) Petri net for the scheme on Fig. 3b

state flags. The first is *incomplete* flag indicating whether there are still some blocks that have not been executed. Then, *hasExecution* flag indicates that there is at least one body block that has been executed during the whole iteration. Finally, *hasPending* flag indicates that there are some blocks being executed at the moment.

One has to mention that the latter flag can be set only in case of assigning a task for execution in asynchronous mode with multiple concurrent executing threads. Assignment of blocks for execution is done by a special component — *Executor*, which determines a strategy of forming such assignments (see sec. III-C).

There are some steps performed by the executing algorithm in each iteration.

1) All three flags are set to false state indicating that no information about state of blocks-to-execute is available yet.
2) An effort to execute a body block is applied to each block contained in the scheme body:
   a) It checks whether the block has already been executed previously. If so, it simply goes to the next block.
   b) It checks whether the block is being executed (in pending state). If so, *hasPending* flag is set, and it goes to the next block.
   c) Finally, it checks whether the block can be executed (has executable state). If so, there is a request for *Executor* to execute the current block (see sec. III-C).
3) If there is at least one block with a state that is different from "executed" (e.g. a block could either be not started at all or be started and still being executed) — that is *incomplete* flag is set up, and there has been at least one block execution, the next iteration is performed.
   The presence of some pending blocks with no block executed is another variation of this case.

The semantics of scheme execution can be represented by an equivalent *system net* [10]. Thus, for the scheme on Fig. 3b containing no choice blocks the equivalent (system) Petri net would appear as on Fig. 4.

C. Model Executor

Model Executor (or just *Executor*) is another important component of the executing subsystem. It is a special agent

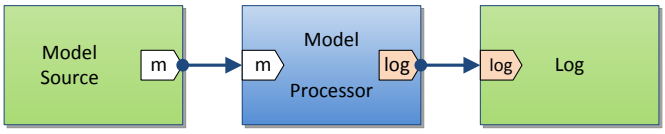


Figure 5. *Pull* (white) and *push* (yellow) ports

linking DPMine workflow model and client software together. Technically, an *Executor* is an object of some class implementing `IExecutor` interface which declares some methods used for executing individual blocks of a workflow model.

Among these methods one can distinguish a couple of the most important ones. The `executeBlock()` method is invoked by some blocks of some special types such as *Bodyness block* whenever there is another block to be executed. The block is passed to `executeBlock()` as a parameter, and the method should be considered as a request from a model to the *Executor* for executing another model's block. The way the *Executor* performs the request is completely determined by the policy implemented in each specific *Executor* class. As an example, one can consider synchronous model of execution. In this case control is not returned to the calling method until the current block is executed or switched to a broken state. Another scenario involves asynchronous tasks assignment to different concurrent threads.

The very first call to `executeBlock()` procedure is performed when the model is being executed, and its *Main Scheme* becomes the very first block passed as a parameter to `executeBlock()`.

Another important method to be implemented by a specific *Executor* class is `execNotify()`. It is used to send to a client application event notifications about the state of a model block being executed. The pointer to the block being processed is the first parameter of the method, and the event notification type is the second. Notification type is a constant from a predefined set including *begin*, *end*, *cancel*, and others. This notification callbacks may be used by the client application for updating its information about the blocks' states.

D. Resource Transferring

Now let us consider the process of transferring resources through the ports of blocks. We distinguish two different approaches for resources transferring: *pulling* and *pushing*. *Pulling* is an approach where a block being executed requests all necessary resources from its input ports which are connected to the corresponding output ports of other "source" blocks. As it was mentioned before, the ability to supply resources for an ahead standing block by the "source" blocks is the main requirement for the current block to be executed. Then, the request for the input ports is redirected to the corresponding output ports through established connectors. Finally, the output ports ask their owner blocks to supply data for the requested resources and give them to the requesting block.

Pushing is another approach illustrated on the Fig. 5. There is a `Log` block having connected to a processing

ModelProcessor block. If executed ModelProcessor generates some log entries (events) and tries to "send" them synchronously through a dedicated output port to all the recipients connected. If a receiving block is ready to get such a "message" it could process it. In the example, the Log block obtains messages from ModelProcessor for displaying and storing.

IV. Model Storage

In the preceding sections we looked at DPMine workflow models from the object model point of view. Now let us consider how a model can be represented as some formal definition in a text-based format.

In the paper [5] we have shown that a model of process mining experiment can be described by using a well-formed XML-based notation. Moreover, this kind of model description was the main form used for importing DPModel objects in ProM.

Having started developing this library we decided to separate the storage subsystem from the core of DPMine library. According to Fig. 1 the storage engine is used for the transformation of an object model to some persistent form such as a text-based file (serialization) and back (deserialization).

As previously, we suppose an XML-based format is one of the best notations for representing a hierarchical irregular structure, which a DPMine model is. In addition, we are now proposing another well-known format with the same XML expression but also which is much more compact and, what is more important, much more appropriate for manual writing: JSON [11].

The main idea is that, as much as the DPMine core system can be extended by developing new block type classes, the storage subsystem can be simultaneously extended to maintain the core extension mirrored. For this very purpose we introduce two special classes: XMLModelLoader and JSONModelLoader. Both of these classes have methods for working with text streams. These streams are used to serialize to and deserialize from a given object model. Some of specific descendants of these classes specify whether a stream is implemented as a file-stream or as another kind of stream.

By processing a file to be loaded as a workflow object model *X*ModelLoader parses the file standard header containing the model description and the model body containing the workflow itself. At a higher level, the model body normally contains only one *Scheme-block* corresponding to the *main scheme*. It means the parsing process should start from parsing this very block.

The extensible nature of (de)serialization mechanism consists in the fact that processing each type of blocks is performed by its dedicated *block loader*, custom *block loaders* for custom block types being added dynamically. The collection of all types of loaders including the standard ones (like *Scheme-block* loader based on *bodyness block* loader, *const block* loader, etc.) is managed by a family of so-called *LoadersFactory* classes existing for each branch of persistent formats: XML, JSON, and so on (Fig. 6).

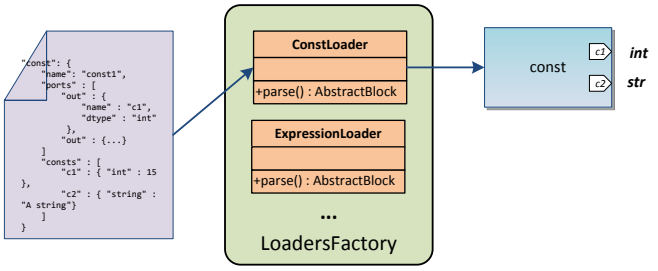


Figure 6. Const loader used for deserialization a JSON-defined "const" block

Registering a new *block loader* for a corresponding block type is done during the library initialization process (normally only once) that uses the underlying collection, which maps the block type name as it appears in the file (as an XML node or a JSON parameter) to a parsing object (normally it is a pointer to a method). When a file loader meets another block description (an XML node or a JSON parameter) it tries to find an appropriate *block loader* and eventually invokes this *block loader* and passes to it the cursor to a file position the block description start is located at. After that, the *block loader* performs the reading of all the necessary data from the stream, constructs a new block object and returns it.

V. Graphical Frontend

Along with the library we supply a GUI application demostrating the ability to integrate DPMine/C library into a GUI client. The application is based on cross-platform Qt library allowing to consider the application as cross-platform. Nevertheless, we are focusing here on a Windows edition to be specific.

The topics to be considered are:

1) How to visualize a workflow model?
2) How to enable user to interact with individual schemes?
3) How to deal with the fact a model can contain custom block types?
4) How to use different look-and-feels?

A. Qt Graphics View Framework

We use Qt Graphics View Framework to make the graphical part of the application. The main components of the framework are the following.

1) QGraphicsView object provides a widget for displaying the contents of a graphic scheme implemented by QGraphicsScene descendant.
2) DPMScheme inherits QGraphicsScene class and adds functionality to handle DPModel specific graphical items in addition to the items handled by its super class.
3) QGraphicsItem is an abstract class for a family of classes representing flowchart shapes — main and miscellaneous.

Flowchart shapes implemented by QGraphicsItem descendants are placed on DPMScheme object, and the latter is visualized by a QGraphicsView container. The goal is to

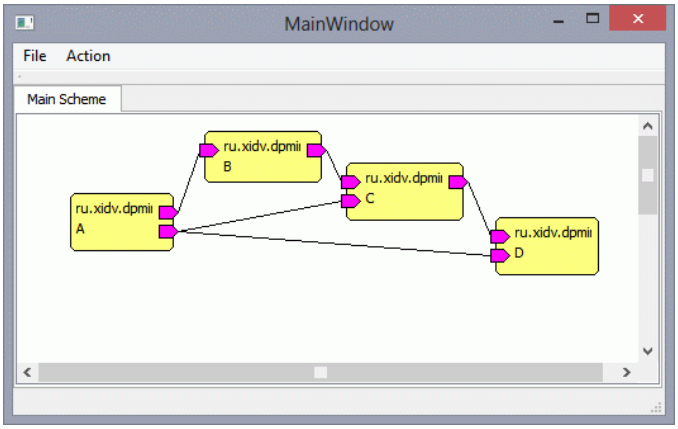


Figure 7. GUI demo application

provide an appropriate graphical renderer for specific block types taking into account the ability to change easily its graphical representation just by changing the renderer. This is the so-called look-and-feel feature.

B. Custom Block Renderers

Just as in the case of custom *block loaders* we propose a dynamic extensible mechanism of *block renderers*. `BlocksRenderers` is the main class supporting a collection of *block renderers*. It contains methods for adding and getting a renderer for a block type given by its type name. Method `getRendererByBlockTypeName` returns an appropriate `BlockTypeRenderer` object used for rendering a given block. If no appropriate `BlockTypeRenderer` for a given block type is found, a default `BlockTypeRenderer` is returned.

`BlockTypeRenderer` is the base abstract class for all the *block renderer* classes. The main method they have to implement is `renderBlock`. It returns a `QGraphicsItem` specifically representing a given block type. This representation can take into account any necessary graphical aspects of a block the developer would like to implement. The base implementation `DefBlocksRender` that can be used for all the block types renders a given block using `DPMDefBlockItem` (a descendant of `QGraphicsItem`). It only shows the block's name and its type as a text label and, of course, renders its ports (Fig. 7).

Block ports are also presented as separate objects of `QGraphicsItem` descendant class (`DPMDefPortItem` is default) grouped by the owner block shape. This is done in order to enable the user to communicate with ports as individual objects.

The presented graphical solution is one of the significant plug-ins for "VTMine framework" (under development) [12].

VI. Conclusion

In this paper we discussed DPMine workflow language and its implementation as a C++ based library. We introduced DPMine main concept and looked at its building elements. Semantics of the model execution has been presented in detail. DPMine block extension approach has been mentioned with regard to the addition of new block types and extension of the storage subsystem and a graphical frontend.

Among the challenges for the future a number of tasks can be identified, namely forming a strong formal semantic system, extending the functionality of DPMine language by introducing some default block types and presenting more complex workflow use cases.

Finally, we have launched a web-site for a DPMine project: https://prj.xiart.ru/projects/dpmine. It is based on a Redmine bug-tracking system and we consider it as a platform for the future DPMine development.

Acknowledgment

The study was implemented in the framework of the Basic Research Program at the National Research University Higher School of Economics (HSE).

References

[1] O. M. G. (OMG), "Business process model and notation (BPMN) version 2.0," Tech. Rep., Jan 2011. [Online]. Available: http://taval.de/publications/BPMN20

[2] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guzar, N. Kartha, C. Liu, R. Khalaf, D. Koenig, M. Marin, V. Mehta, S. Thatte, D. Rijn, P. Yendluri, and A. Yiu, "Web Services Business Process Execution Language Version 2.0 (OASIS Standard)," WS-BPEL TC OASIS, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, 2007.

[3] W. M. P. van der Aalst, "What makes a good process model? - lessons learned from process mining." *Software and System Modeling*, vol. 11, no. 4, pp. 557–569, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/sosym/sosym11.html#Aalst12

[4] S. Shershakov, "DPMine: modeling and process mining tool," in *Proceedings of the 7th Spring/Summer Young Researchers' Colloquium on Software Engineering, SYRCoSE 2013*, 2013.

[5] ——, "DPMine/P: modeling and process mining language and ProM plug-ins," in *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, A. N. Terekhov and M. Tsepkov, Eds. ACM New York, NY, USA, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2556622&CFID=415147702&CFTOKEN=35395117

[6] *Web Services Business Process Execution Language Version 2.0*, OASIS Std.

[7] W. M. P. van der Aalst, "Three good reasons for using a Petri-net-based workflow management system." in *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96), Cambridge, Massachusetts, Nov. 14-15, 1996*, Navathe, S. and Wakayama, T., Eds., 1996, pp. 179–201.

[8] W. M. P. van der Aalst and A. H. M. ter Hofstede, "YAWL: Yet another workflow language," *Inf. Syst.*, vol. 30, no. 4, pp. 245–275, Jun. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.is.2004.02.002

[9] S. Jablonski and C. Bussler, *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.

[10] W. M. P. van der Aalst, "Decomposing Petri nets for process mining. A generic approach," 2012.

[11] The application/json media type for JavaScript object notation (JSON). [Online]. Available: http://tools.ietf.org/html/rfc4627

[12] P. Kim, O. Bulanov, and S. Shershakov, "Component-based VTMine/C framework: Not only modelling," 2014, in press.