Business Process Modeling in Object-Oriented Declarative Workflow

Marcin Dąbrowski, Michał Drabik, Mariusz Trzaska, Kazimierz Subieta
Polish-Japanese Institute of Information Technology
Warsaw, Poland
{mdabrowski, mdrabik, mtrzaska, subieta}@pjwstk.edu.pl

***Abstract*—The paper presents motivations, the idea and design of an object-oriented declarative workflow management system. The main features that differ this system from many similar systems are: inherent parallelism of all workflow instances and tasks, the possibility of dynamic changes of running process instances and integration of workflow instances with an object-oriented database. Workflow instances, tasks, subtasks, etc., are implemented as so-called active objects, which are persistent data structures that can be queried and managed according to the syntax and semantics of a query language. and also possess active parts that are executable. The prototype has been implemented on the basis of ODRA, an object-oriented distributed database management system. As the workflow programming language we use SBQL, an object-oriented database query and programming language developed for ODRA.**

Keywords-workflow; object-oriented; declarative; query language; active object, dynamic workflow change; ODRA; SBQL

I. INTRODUCTION

Current workflow technologies, developed mainly by commercial companies and standardization bodies, see for instance [1, 2, 3], present a considerably well recognized domain, with a lot of commercial successes. The core of the current approaches to workflows is the control flow graph, which determines the order of tasks performed by a particular process instance. Other issues related to workflows, such as resource management, workflow participant assignments, database structure and organization, transaction processing, synchronization of parallel activities, exception handling, tracking and monitoring of workflow processes, are frequently treated with attention, but are seen as secondary with respect to the work control flow. The model based on a control flow graph is defined formally as a Petri net [4].

There are problems that undermine applications of workflow management systems in important business domains. Below, we list the following features that are frequently required in complex business applications, but are absent or poorly supported by workflow systems:

1. Mass parallelism of tasks within workflow processes.
2. Dynamic changes of workflow instances during their run.
3. Reactions to unexpected events or exceptions and aborting running processes or their parts.
4. Resource management as a main workflow driving factor.

Below, we discuss the above aspects.

Ad.1. Currently workflow systems enable parallel sub-processes through splits and joins (AND, OR, XOR) that are explicitly determined by the programmer. Such a form of parallelism is insufficient for many business cases. During the run, a process instance could be split into many parallel sub-processes, but their number is large and unknown during development of the process definition. For example, processing an invoice requires splitting it into as many sub-processes as the items that the invoice consists of. This typical situation cannot be covered by explicit splits and joins. Moreover, as noted by Reichert and Dadam [5], it is often not convenient and not efficient to determine task sequences in advance.

Ad2. Although there is a valuable research (e.g., [5, 6, 7, 8, 9, 10, 11]) aiming at dynamic changes of process instances, especially workflow patterns [12, 13], it can be anticipated that the scope of the changes must be limited. There are several problems with modifications of a currently executed process instance graph:

- Current workflow programming languages are not prepared to deal with dynamic changes of a running code.
- Parts of a flow control graph have no identity, they cannot be separated from other parts and they are not described by some metamodel (like a database schema).
- Process instance graph elements are tightly interconnected. If one would try to alter the code (e.g. by removing some its part) the problem is how to fix other elements to create a consistent whole.
- Changes can violate the consistency of process instances, hence some discipline of changes is required.
- If many possible actors are allowed to alter a process instance graph, then elements of the graph should follow ACID transactions.

Ad3. Usually, programming languages have programming means to define and process exceptions (events). However, this concerns only situations when exceptions are known during developing a process definition. The behavior of business processes is frequently unpredictable. There are exceptional situations that are known only at the time when process instances are already running. For instance, a new type of malicious attack on a banking workflow system is discovered in situations when

 ISBN: 978-1-61208-230-1

thousands of process instances are already executed. The workflow system administrator has practically the only option: abort processes, change the process definition and start processes from the beginning. From the business point of view this could be unacceptable and might generate a huge additional cost. Aborting already running processes is a problem, because they engage entities and resources from the business environment (clients, personnel, documents, contracts, etc.). They may require a lot of compensating actions, which must be done manually, with no help from the workflow system.

Ad4. In currently developed workflows, the work control flow (a la Petri net) is on the primary plan and the resources (people, money, time, work power, equipment, infrastructure, offices, vehicles, etc.) are secondary and sometimes not taken into account at all. This is unnatural for business processes because just availability, unavailability of resources and their monitoring, planning and anticipating are the main factors that determine a process control flow. Just availability of resources should trigger some tasks. Because information on resources is usually a property of a database supporting the workflow system, conditions within a workflow control flow graph should include accesses to a database. This is usually impossible in typical workflow programming languages or burdened by an infamous effect known as impedance mismatch [14] between querying and programming.

The above issues were the reasons that we started the research on a new workflow management system that will be able to overcome limitations of the current systems concerning mass parallelism and dynamic changes of running workflow instances. The assumptions of our design is that an element of a workflow instance should have a double nature. On the one hand, it should be perceived as a data structure (an object) that can be addressed by a database query and programming language. The structure is to be stored in a database and should be the subject of database transactions. On the other hand, the element should contain executable parts, i.e., the code of a workflow process or sub-process.

This way, we have come to the concept of active object. An active object is a database object that contains some static parts (attributes) and some active parts (codes). We distinguish four such parts: *firecondition, executioncode, endcondition* and *endcode*. An active object waits for execution until the time when its *firecondition* becomes true. After that, the object's *executioncode* is executed. The execution is terminated when either all the actions are completed (including actions of active sub-objects) or its *endcondition* becomes true. After fulfillment of the *endcondition* some terminating actions can be executed through *endcode*. This may be required to terminating some actions, e.g. closing connections, aborting transactions, setting a new object state, etc. Active objects belong to their classes, follow the principle of encapsulation and are typechecked according to the strong typing system. They can be updated as regular database objects. Preventing undesired updates can be accomplished by well-known database capabilities such as user rights, integrity constraints, triggers and active (business) rules. Unexpected events can be served by inserting new active sub-objects into running active objects and/or by altering active objects.

Active objects accomplish an important feature: mass parallelism of executed tasks. In principle, all active objects act in parallel. In life, tasks performed by people can be done in parallel with no conceptual limitations. Some tasks, however, must wait for completing other tasks and this model can be expressed as a PERT (Program Evaluation and Review Technique) graph. Active objects act as PERT graphs: if object A has to wait for object B, then the *firecondition* of A tests the state of B, which should be set to "completed" when B is terminated. This way, one can determine the sequence of processes, but this does not constraints one from using parallelism whenever possible. Because the sequence of tasks is not determined explicitly, we describe this workflow model as "declarative". Note that this idea of declarative workflow processes is considerably different from the idea of the DECLARE model presented in [15], which is a logic-oriented formalism for specification of various dependencies between (sequences of) events. In our case, "declarative" means that the programmer specifies a workflow code as collections of (nested) active objects, with *fireconditions* and *endconditions* specified by means of the declarative query language SBQL[16].

In our idea, workflow resources, as any database properties, can be used to form *fireconditions* and *endconditions*. In this way the resource management is properly shifted on the first plan. Both active objects and resource description objects are integrated in the same database thus can be addressed by the same integrated query and programming language. Hence, any form of impedance mismatch is avoided.

The widely recognized paper devoted to dynamic workflow changes is [5]. It presents some framework for formalizing process graphs and updating operations addressing such a graph. There are valuable observations concerning the necessity of dynamic workflow changes for real business processes and the necessity of strong discipline within the changes to avoid violation the consistency of the processes. Numerous authors follow the ideas of this paper. The fundamental difference of our approach is that the process control flow graph is not explicitly determined. It can be different from one run to next run, depending on the state of the workflow environment, database, computer environment, *fireconditions* and *endconditions*. The problem of the necessity of various control flow graphs for the same business process is one of the motivations for the research presented in [5], but it is not easy to see how such a feature can be achieved within the proposed formal workflow model. In our idea the feature is an inherent property.

Some disadvantage of our concept concerns the performance. Decreasing performance can be caused by late binding and the necessity of cyclic checking of *fireconditions* and *endconditions*. We believe that performance problems of our idea can also be overcome by new optimization methods and new computer architectures.

The prototype is implemented under the ODRA system [17]. As a workflow programming language, we use SBQL,

 ISBN: 978-1-61208-230-1

an object-oriented query and programming language developed for ODRA.

The paper is organized as follows. Section 2 presents the concept of an active object. Section 3 describes the implemented prototype. Section 4 presents a comprehensive example of a declarative workflow. Section 5 concludes the paper.

II. ACTIVE OBJECTS

In the following, we use the term *active object* as a generalization of process instance and task instance. Because of the relativity of objects assumed in SBQL, components of active objects are active objects too. Due to this, there is usually no need to distinguish between process instances and task instances – all are represented as active objects. To represent process and task instances, active objects are specialized, and belong to a special class named *ActiveObjectClass*, which contains basic typing information, basic methods and other necessary invariants.

An active object is a nested object with the following main properties:

- Unique internal object identifier.
- External (business) name that can be used in source programs.
- Certain number of public and private attributes.
- One distinguished attribute (sub-object) containing an SBQL procedural workflow *executioncode* of a process or a task; it may contain an empty instruction only.
- One distinguished attribute (sub-object) containing an SBQL code with a *firecondition* (a condition for starting the run of the given active object).
- One distinguished attribute (sub-object) containing an SBQL code with an *endcondition* (a condition for terminating the run of the given active object). An *endcondition* may be absent. In this case the action of an active object is terminated when its *executioncode* is terminated and/or when all its active subobjects are terminated.
- One distinguished attribute (sub-object) containing an SBQL code with an *endcode* (a code executed to consistently terminate the run of the given active object). An *endcode* can be absent.
- Any number of named pointer links (binary relationship instances) to other (active or passive) objects.
- Any number of inheritance links connecting the given object to its classes (multiple inheritance is supported).
- Any number of nested active objects. The construction of a nested active object is identical to that of a regular active object (the object relativism is supported). The number of nesting levels for active objects is unlimited.

When an active object consists of active sub-objects, the *endcondition* determines whether the process or task is completed. An *endcondition* can accomplish all kinds of joins (AND, XOR) of parallel processes, and much more.

For example, if within an active object *Invoice* there are many (unknown number of) active sub-objects *TestingAnItem*, then we can impose the *endcondition* of *Invoice* (the end of the invoice checking process) in the form of a query involving an universal quantifier:

```
forall TestingAnItem as x (x.State =
"completed" or x.State = "cancelled")
```

We can also impose more complex conditions. For instance, let the cost of an invoice item will be stored within *TestingAnItem* as a *Cost* attribute, and assume that the entire invoice is checked if more than 95% of its total cost is checked. In this case, the endcondition will have the form:

```
sum((TestingAnItem where State =
"completed").Cost) / Invoice.TotalCost > 0.95
```

Because active objects are regular objects in the SBQL terms, they can be manipulated without limitations. For instance, active objects can be altered and deleted. Their state can be changed, including the code of active parts. New active sub-objects can be inserted into an active objects. This feature makes it possible to split the process (represented by the active object) into any number of subprocesses (inserted active subobjects). Proper construction of the object's *endcondition* (e.g., with the use of quantifiers) makes it possible to do any join of them, as illustrated in examples.

Active objects are specified by their classes and follow the strong typechecking. The definition of a workflow process determined by its class can be instantiated by creating an active object of this class. The object creation follows the standard routine of SBQL. The only difference concerns executable parts: during instantiation their codes are created as strings and then compiled to bytecodes.

III. DESIGN AND IMPLEMENTATION OF THE PROTOTYPE

The implemented prototype makes it possible:

- Creating and modifying workflow definitions;
- Instantiating them (creating workflow instances);
- Running a workflow monitor which processes workflow instances.

The prototype has been implemented as a web application using several technologies. The web part utilizes Groovy [18], Grails [19] and JavaScript [20]. The ODRA DBMS, the ODRA wrapper and the process monitor are written in Java.

The main part of the system resides on the Tomcat [21] servlet container hosting most of the application logic. The most important parts are the following:

- The module for generating GUI. It is based on the core Grails framework technology called GSP (Groovy Server Pages). It is similar to well-known JSP (Java Server Pages);
- The application logic which manages the workflow model on the functional level. It provides an interface to administrative tasks in a workflow system for all applications. It is suitable not only for our custom built GUI interface, but also for any Java-based application.
- The ODRA wrapper simplifies all tasks related to the ODRA DBMS. ODRA is responsible for storing workflow related information (definitions, instances) and executing SBQL codes within active objects.

 ISBN: 978-1-61208-230-1

- The process monitor accomplishes time sharing among active parts. It periodically switches the control flow among all currently executed parts of active objects and their subobjects.
- The cache memory speeds up the access to commonly utilized DB objects.

The client component is executed in the standard-compliant web browser and consists of the following features:

- Regular web forms which are used for creating and updating instances and definitions.
- An AJAX part written in JavaScript using the jQuery library. Such an approach makes it possible to use powerful widgets like definitions/instances trees (Fig. 1) or SBQL code editor with syntax highlighting. Another advantage was lack of reloading a web page (post/get) in some cases, i.e., auto refreshing of instances' status in the tree. As a result overall user experience was greatly enhanced.

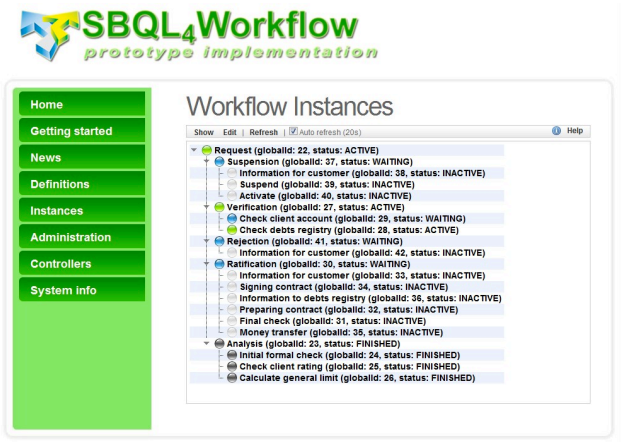

Figure 1. A workflow instances tree

The last two remaining architecture's items are: the ODRA system and a mail server. The last one is utilized for sending progress messages to parties involved in a workflow.

The schema of a database used to store workflow data, is presented in the Fig. 2. The process objects represent structures created by the workflow programmer before it is actually ran. Once a process is initiated, all data, including the data of sub-processes, is copied to the corresponding ProcessInstance objects. The parent-child bidirectional pointer, combined with SBQL query operators, gives a great flexibility in expressing conditions and codes. For instance:

- Find all my children (the code is written with regard to one particular *ProcessDefinition*).
- Find my parent.
- Find a process with a given status.
- Find a process with a given name.

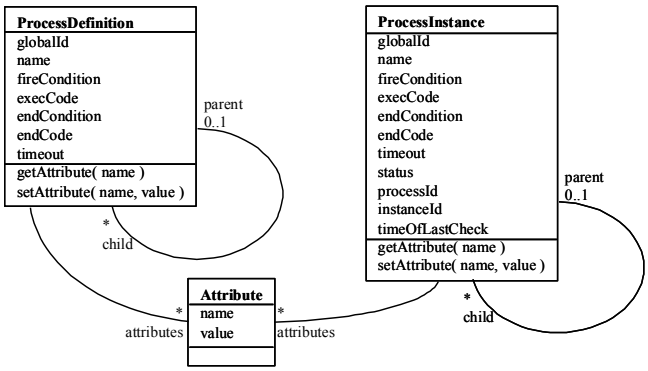

Figure 2. ODRA database schema

These constructs can be easily combined for more complex search, for instance:

- Find a child that has a certain name and status.
- Check if all my children have the status 'Finished'.
- Find my "brother" (using parent.children).
- Find all my "nephews" (using parent.children.children).

To allow processes to store "ad-hoc" some additional data we have provided the Attribute class with a set of methods in the *ProcessDefinition* and *ProcessInstance* classes. Attributes can be easily used to control the flow (when the conditions are based on them) and enable the communication between processes (as one process can query other process attributes and can change their values).

The *ProcessMonitor* is a Java based application, that can be run as a separate thread on a separate machine. Its duty is to periodically check (basing on timeouts) each *ProcessInstance*. Then, according to the values retrieved from condition codes, the *ProcessMonitor* executes the inner code of the process and pushes it forward through the workflow.

IV. SAMPLE DECLARATIVE WORKFLOW DEFINITION

As an illustration, we have created a sample workflow, which utilizes basic concepts of our idea. The workflow application supports processing of a credit request within a bank. It is a complex structure of active objects representing various tasks. The structure is presented in Fig. 3. Apart from objects representing processes, there are resource objects that are available through names such as *Customer*, *ApplicationForm*, *Account* and *Contract*. A rough scenario for the *Request* process is described below.

1. A customer submits an application for a credit in the form of an *ApplicationForm* object.
2. After checking that all of necessary resources are available the *Analysis* sub-process is activated.
3. The data is checked formally by the analyst for formal and business correctness (Initial formal check).
4. If the data is incorrect, the customer is informed about that and further processing of the application is suspended (Suspension) until reaction of the customer is received. If there is no reaction the application is rejected, and the customer is informed about that by an appropriate e-mail message (Rejection).

 ISBN: 978-1-61208-230-1

5. If the data is correct the client rating is calculated (Calculate Client Rating).
6. After successful calculating of the client rating, a check is performed if the amount of the credit does not exceed the general bank limit (Calculate general limit).
7. A positive result of the Analysis sub-process activates the Verification sub-process.
8. Verification consists of two stages:
   a. Checking if the customer is not present in the government registry of persons having debts;
   b. Checking if the customer has an account within the bank; if not, creating such an account.
9. If this sub-process is successfully completed, the sub-process Ratification is triggered.
10. The sub-process Ratification is split into sub-processes:
    a. Checking if the customer's current income is sufficient for the requested credit (Final check).
    b. Preparing contract for the customer (Preparing contract);
    c. Sending information to the customer (Information to customer);
    d. Signing the contract with the customer (Signing contract);
    e. Transfer of the money to the customer's account (Money transfer);
    f. Checking and sending information to the government registry of customers that apply for credits (to avoid many applications of the same customer to different banks submitted at the same time).
11. If these tasks are completed (successfully for the customer or not), the process instance is terminated.
12. If at any stage the application is rejected the appropriate information is sent to the customer.

Let us consider the *Ratification* sub-process in more detail. It consists of six sub-processes: *Final check, Preparing contract, Information for customer, Signing contract, Money transfer* and *Information* to debts registry. In order to start the *Ratification* process the fire condition should check if the parent's attribute 'state' is empty and the Verification process has got finished status. This condition means that the application has not been rejected yet and the Verification sub-process is finished. After satisfying the fire condition, *Ratification* process changes its status to *Active* and all of its children changes status to *Waiting*. A *Ratification* process is ended when all its children are completed or when the application is rejected.

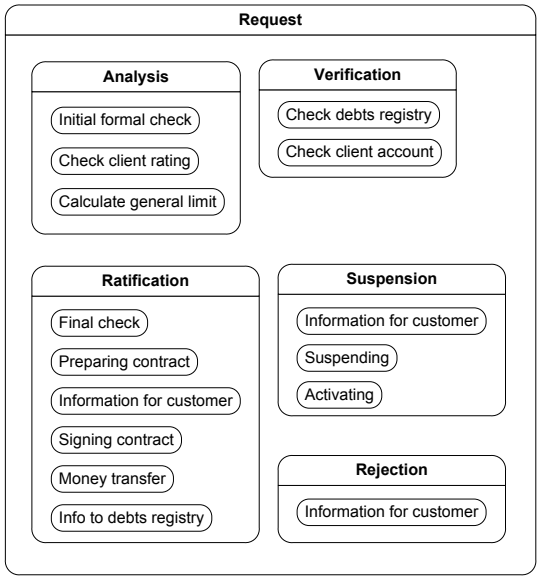

Figure 3. Structure of the Request process

When the *Ratification* is active, the fire condition of the child with the name *Final check* is checked. It fires as soon as its parent has got active status. When the process activates, the code of this task is executed. The purpose of this code is to check if the customer can afford such a credit and according to that sets the proper value to the attribute "state" of the *Request* object. The *endcode* of the *Final check* is absent, hence the process ends immediately after completing the execution code.

The next process in order is *Preparing contract*. It fires as soon as *Final check* is finished and the state attribute of a *Request* process has the "accepted" value. The purpose of this process is to create a new *Contract* object assigned to an application form filled by the customer with start date equal to the current date and with an attachment being a reference to the application form of the customer. If the contract has been successfully created the process ends.

Finishing of *Preparing contract* process activates *Information for customer*. The main task of this process is to send an e-mail to the customer with the information that the contract is ready to sign up. Depending on the result of this operation the attribute *mailSent* is set with a proper value. If sending does not succeed, the status of the process is changed to *Waiting*, so the next process monitor check will trigger its run again. When the e-mail is sent the process ends. After informing the customer on the contract, the processing waits for the signature. The next process *Signing contract* provides information if a contract has been already signed or not. It is started after finishing *Preparing contract* and is active till a *contractSigned* attribute is false.

When the contract is signed, the bank transfers the money into the customer account. The *Money transfer* process is responsible for this action. It is activated when the *Signing contract* process is finished.

The execution code for this process updates the amount attribute from the customer's *Account* object with the value

 ISBN: 978-1-61208-230-1

of the *creditAmount* attribute from the specific *ApplicationForm* object. The process ends immediately after completing this action (no *endcondition*).

The last action in the ratification procedure is sending an info about a customer to a debts registry. After completing all the sub-processes the *Ratification* process is finished.

The manager of workflow processes can do any changes to process instances, including currently running instances by simple database updates. For instance, for any reason he/she can delete active object *Check client rating* from active object *Analysis* for the given customer *Request*. It is possible that in such a case the *endcondition* of the *Analysis* object should be changed too.

V. CONCLUSION AND FUTURE WORK

The perception of workflow processes as autonomous objects can be very useful in terms of maintaining and managing process definitions and execution. Controlling the process execution with fire and end conditions gives a workflow creator a powerful tool to create very flexible and advanced control structures. Moreover every process attribute such as conditions, execution code etc. can be accessed in every moment of the process lifetime, which gives the opportunity to apply a changes to an already working workflow instance if needed. The mentioned features had been successfully implemented in working prototype. It gives a foundation to achieve important features like mass parallelism and flexible resource management.

The idea is very new, hence it presents a lot of opportunities for future research. One of the research lines concerns mass parallelism of processes and tasks executed on many (thousands of) servers. This require developing and implementing a process monitor and a task balancing tool. Another research concerns a user-friendly API for dynamic process changes. Proper modifications of notations such as BPMN (Business Process Modeling Notation) [22] and execution languages such as XPDL (XML Process Definition Language) [23] and BPEL (Business Process Execution Language) [24] could also be the subject of research. There is also a need for preventing running processes from undesired changes using such means as user rights, semi-strong type checking, triggers and business rules.

REFERENCES

[1] IBM developer works: Business Process Execution Language for Web Services, ver. 1.1, May 2003.

[2] OMG. Business Process Modeling Notation (BPMN) specification. Final Adopted Specification. Technical Report, 2006

[3] WfMC, WorkFlow process definition interface – XML Process Definition Language. WfMC TC 1025 (2.1); October, 10, 2008

[4] Petri nets: http://www.petrinets.info/

[5] M. Reichert and P. Dadam. ADAPTflex: Supporting dynamic changes of workflow without loosing control. Journal of Intelligent Information Systems, 10(2), pp. 93-129, 1998

[6] C.A.Ellis. K.Keddara, GRozenberg. Dynamic change within workflow systems. Proc. ACM Conf. on Organisational Computing Systems (COOCS 95)

[7] C.A.Ellis. K.Keddara, and J.Wainer. Modelling workflow dynamic changes using time hybrid flow. In Workflow Management: Net based Concepts, Models, Techniques and Tools (WFM'98), 98(7), Computing Science Reports, pp. 109-128. Eindhoven University of Technology, 1998

[8] D.C.Ma, J.Y.-C.Lin, M.E.Orlowska. Automatic merging of work items in business process management systems. Proc. 10th Intl. Conf. on Business Information Systems (BIS2007), Poznań, Poland, 2007

[9] W.M.P. van der Aalst. Generic workflow models: How to handle dynamic change and capture management information? Proc. 4th Intl. Conf. on Cooperative Information Systems (CoopIS'99), Los Alamitos, CA, 1999

[10] S.Sadiq, O.Marjanovic, M.E.Orlowska. Managing change and time in dynamic workflow processes. Intl. Journal of Cooperative Information Systems (IJCIS), 9(1-2), 2000

[11] S.Sadiq, M.E.Orlowska. Architectural considerations in systems supporting dynamic workflow modification. Proc. 11th Conf. on Advanced Information Systems Engineering, CAiSE99, Heidelberg, Germany, 1999

[12] W.M.P. van der Aalst, A.H.M.Hofstede, B.Kiepuszewski, A.P.Barros. Workflow patterns. Distributed and Parallel Databases, 14(3), pp. 5-51, 2003

[13] G. Vossen, M. Weske: The WASA Approach to Workflow Management for Scientific Applications . In: Workflow Management Systems and Interoperability. ASI NATO Series, Series F: Computer and Systems Sciences, Vol. 164, pp. 145-164. Berlin: Springer 1998

[14] Impedance mismatch: http://www.sbql.pl/Topics/ImpedanceMismatch.html

[15] F. M. Maggi, A. J. Mooij, and W. M. P. van der Aalst, User-Guided Discovery of Declarative Process Models , 2011 IEEE Symposium on Computational Intelligence and Data Mining, 2011

[16] SBQL: Stack-Based Query Language: http://www.sbql.pl/

[17] ODRA: Description and Programmer Manual. http://www.sbql.pl/various/ODRA/ODRA_manual.html, 2008

[18] Groovy: A dynamic language for the Java Platform. http://groovy.codehaus.org/

[19] Grails: http://grails.org/

[20] Javascript: http://www.w3schools.com/js/default.asp

[21] Apache Tomcat: http://tomcat.apache.org/

[22] BPMN: Business Process Modeling Notation: http://www.bpmn.org/

[23] XPDL: XML Process Definition Language: http://www.wfmc.org/xpdl.html

[24] BPEL: Business Process Execution Language: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

 ISBN: 978-1-61208-230-1