Dynamic Changes of Workflow Processes*

Marcin Dąbrowski[1], Michał Drabik[1], Mariusz Trzaska[1], Kazimierz Subieta[1,2]

[1)] Polish-Japanese Institute of Information Technology
{mdabrowski, mdrabik, mtrzaska, subieta}@pjwstk.edu.pl
[2)] Institute of Computer Science Polish Academy of Sciences

Abstract. The paper presents motivations and technical assumptions of a prototype object-oriented declarative workflow management system that is developed and implemented at the Polish-Japanese Institute of Information Technology in Warsaw. The main assumption that differs this system from many similar systems is the possibility of dynamic changes of running process instances. The workflow environment consists of so-called active objects, which play a double role. Active objects are persistent data structures that can be queried and managed according to the syntax and semantics of a query language. On the other hand, active objects possess active parts that are executable. 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 execution code is executed. The execution of a given active object is terminated when either all the actions are completed or its endcondition becomes true. After fulfillment of the endcondition some terminating actions can be executed through endcode. In principle, all active objects are executed in parallel, which amounts to a fundamentally different attitude to the control flow. The flow is determined dynamically using only fireconditions and endconditions that depend on a workflow environment state, a database state, a machine state, a state of resources, etc. There are several big advantages of this approach to the workflow, in particular, parallel execution of many processes and their synchronization are not constrained and the workflow environment can be the subject to ad hoc changes. 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

1. Introduction

Current workflow technologies, developed mainly by commercial companies and standardization bodies, see for instance [1, 2, 3, 4], present a quite well recognized

* This research is supported by the Polish Ministry of Science and Higher Education through the grant N N516 3755 34.

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 planning and 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 the control flow graph is defined more formally as a Petri net, and such a net is able to specify – to some extent – synchronization of the parallel tasks performed by a particular workflow process instance.

Nevertheless, there are still 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:

- Dynamic changes in workflow instances during their run. A dynamic change can be performed automatically, as a result of some situation in the entire workflow environment, or as a consequence of the state of a particular process. A change can also be performed ad hoc and manually by some external agent, for instance, by the workflow system administrator.
- Parallel execution of tasks within workflow processes. The current systems are based on particular splits and joins that are explicitly determined by the programmer, but in many cases such parallelism is insufficient. For instance, it may happen that during a run of a process instance the process should be split into many parallel sub-processes, but their number is large and unknown for the process definition.
- Parallel execution of sub-processes on independent hardware. Actually, little or no support for this feature is assumed by the process instance control flow graph.
- Aborting a process or some of its parts. Aborting can result from some situations that are recognized in the entire workflow environment. For instance, if from the environment analysis it is known that some resources (e.g. money) have been exhausted, then it makes no sense to continue some workflow processes, because they cannot be completed anyway. Currently, such situations are handled manually, with a lot of possibilities for non-optimal human intervention.
- Resource management. Workflows deal with various resources, in particular, workflow performers, money, time, computer and office infrastructure, specialized equipment, machines, vehicles, etc. In current approaches to workflows the resource management is secondary and driven by the control flow. This is unnatural for many business processes, because just the availability or unavailability of resources is a natural factor which can trigger or suspend some tasks or processes.
- Tracking and monitoring. Tracking and monitoring concerns the entire workflow environment and all process instances. This feature means that each individual process instance can be the subject of tracking concerning all the past operations that it has executed, concerning the current operation, and concerning the anticipated further operations. Monitoring concerns the entire population of workflow process instances, and involves asking analytical queries about the whole population. Tracking and monitoring require organizing the entire workflow

environment and all process instances as a data structure with a well developed schema, so querying it via a query language is possible.

- Transaction processing. Usually, in this context the main problem tackled in the literature concerns long-living transactions that last weeks or months, and thus make some data unavailable for other instances. There are, however, other peculiarities of transaction processing in workflow processes. In particular, some parts of transactions that concern human action cannot be reversed. In such a case, the transaction should be augmented by a compensating transaction. Distributed transactions concerning workflows may need more sophisticated algorithms than 2PC or 3PC. There are also cases when the isolation axiom of transaction processing must be violated. For instance, some workflow transactions may need information which transactions have locked the currently required resources.
- Integrated object-oriented model and query/programming language. Currently, workflows are supported by relational databases, as a rule. This causes various forms of impedance or culture mismatch. In particular, an object-oriented workflow application schema must be converted to a relational schema, which makes programming and program maintenance much more complex.

After analysis of the current state-of-the-art in the domain we have concluded that classical approaches to workflows based on definitions of processes in the form of control flow graphs (Petri nets) have unavoidable conceptual limitations that disallow to achieve the mentioned qualities. Although there is a substantial research aiming at dynamic changes of process instances, see [5, 6, 7, 8, 9, 10, 11, 12], it can be anticipated that their scope must be limited. There are at least three problems with modifications of a currently executed process instance graph. (1) Parts of the graph have no identity, they cannot be separated from other parts and they are not described by some metamodel (like a database schema). The existence of a such a metamodel is a sine-qua-non condition for user's awareness concerning what and how to modify. (2) A process instance graph elements are tightly interconnected. In languages with early binding any actions on a running program code are impossible by definition. Assuming late binding, if one will try to alter the code (e.g. to remove some element) the problem is how to "sew up" other elements to create a consistent whole. (3) If many possible actors are allowed to alter a process instance graph, then elements of the graph should follow the discipline of ACID transactions. It can also be necessary for a single actor, because changing a process instance element may require changing other elements in the environment (in particular, a database). To avoid inconsistency, in case of impossibility of doing some action, the transaction should be aborted and all the actions that belong to it should be rolled back.

The above issues were the reason that we started the research on a new workflow management system that will be able to overcome limitations of the current systems concerning dynamic changes of running workflow instances. The obvious 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 an executable part, i.e. the code of a workflow

sub-process. A consequence of this assumption is that our system should follow late rather than early binding.

In 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.

In our intention, active objects accomplish another important property that is poorly designed in classical approaches: mass parallelism of executed tasks. In life, work 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 graph. In workflow processes the parallelism is achieved by splits and joins (AND, XOR) that are explicitly determined by the programmer. Splits and joins are a very primitive form of synchronizing parallel processes insufficient for many business situations, for instance, when the number of parallel subprocesses is unknown during preparation of a process definition.

Hence we have assumed that in principle all active objects act in parallel. As in 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. In this way one can determine the sequence of processes, but this does not constraints one from using parallelism whenever possible.

Another feature that is poorly covered by traditional approaches to workflows concerns the resource management. If under the term "resource" we understand workflow participants, professionals, documents, time, money, offices, equipment, wehicles, etc. then the whole workflow domain can be described as the resource management. In current approaches, the resource management is on the secondary plan, subordinated to the workflow control flow. But just availability or unavailability of resources, planning of them, anticipating of their use and presence should cause triggering of some tasks. It makes little sense to trigger a task "checking the document by the lawyer" if a competent lawyer is unavailable. The state of resources is recorded in a database. Lack of access to it by workflow processes causes the situation that some processes cannot be continued due to the lack of resources. For instance, if completing the entire process requires tasks A and B to be executed in parallel, but B cannot be continued for lack of resources, then A should be immediately terminated too. This is a difficult situation for the traditional workflow systems, requiring ad hoc human intervention.

In our idea, resources, as any database properties, can be used to form fireconditions and endconditions. In this way the resource management can be properly shifted on the first plan, to avoid the above mentioned problems.

We are aware of some disadvantage of our concept: performance. Decreasing performance can be caused by late binding and the necessity of cyclic checking of

fireconditions and endconditions. On the other hand, the performance can be improved by new possibilities of parallel processing on many servers. Performance is a favorite argument of conservative radicals advocating the current state in a domain. Our answer is based on analogy. When the relational model in databases was proposed, the major argument of the opposition was possible performance problems. It appears that due to query optimization this argument was compromised. We believe that performance problems of our idea can also be overcome by new optimization methods and new computer architectures.

The description of the idea of declarative object-oriented workflow is thoroughly described in the monograph [13]. The prototype is implemented under the ODRA system [14]. As a workflow programming language we use SBQL [15, 16, 17], an object-oriented query and programming language developed for ODRA.

The rest of the paper is organized as follows. In section 2 we describe 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.

2. Active Objects

In the following we will use the term *active object* as a generalization of *process instance* and *task instance*. Because of the relativity of objects assumed in SBA and 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. As ActiveObjectClass does not contain object names as invariants, each active object can have its own business name, and may belong to its own class.

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

1. Unique internal object **identifier**.
2. External (business) **name** that can be used in source programs.
3. Certain number of public and private **attributes**.
4. One distinguished attribute (sub-object) containing an SBQL procedural workflow **executioncode** (compiled to a bytecode) of a process or a task. An execode may contain an empty instruction only.
5. One distinguished attribute (sub-object) containing an SBQL code (compiled to a bytecode) with a **firecondition** (a condition for starting the run of the given active object).
6. One distinguished attribute (sub-object) containing an SBQL code (compiled to a bytecode) 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.
7. One distinguished attribute (sub-object) containing an SBQL code (compiled to a bytecode) with an **endcode** (a code executed to consistently terminate the run of the given active object). An endcode can be absent.

8. Any number of named **pointer links** (binary relationship instances) to other active or passive objects.
9. Any number of **inheritance links** connecting the given object to its classes (multiple inheritance is supported).
10. 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.

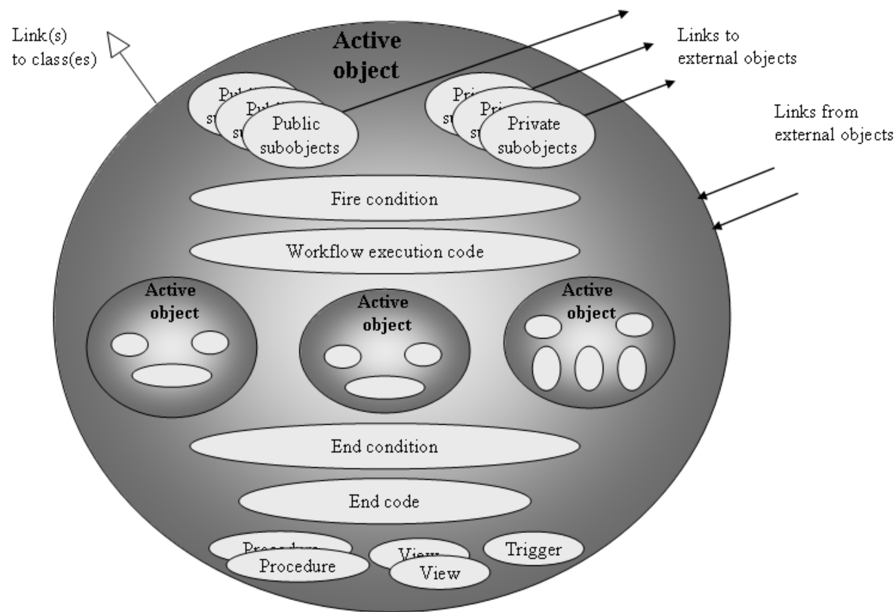

Fig.1. Active object

Fig.1 illustrates the construction of an active object. In case of an active object that 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”)

Of course, 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 the above examples.

Active objects are specified by their classes and follow the semi-strong typechecking mechanism [18]. Too strong typing system may decrease the flexibility of possible dynamic changes of workflow instances. However, we cannot switch off the typechecking of SBQL for two reasons: discovering typing errors during compilation of queries/programs and (even more important) in SBQL typechecking is a prerequisite for query optimization.

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 active parts: during instantiation their codes are created as strings and then compiled to bytecodes.

3. 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 a couple of technologies and software frameworks, Fig.2. The web part utilizes the Groovy, Grails and JavaScript. The rest (the ODRA DBMS, the ODRA wrapper and the process monitor) is written in Java with some other libraries.

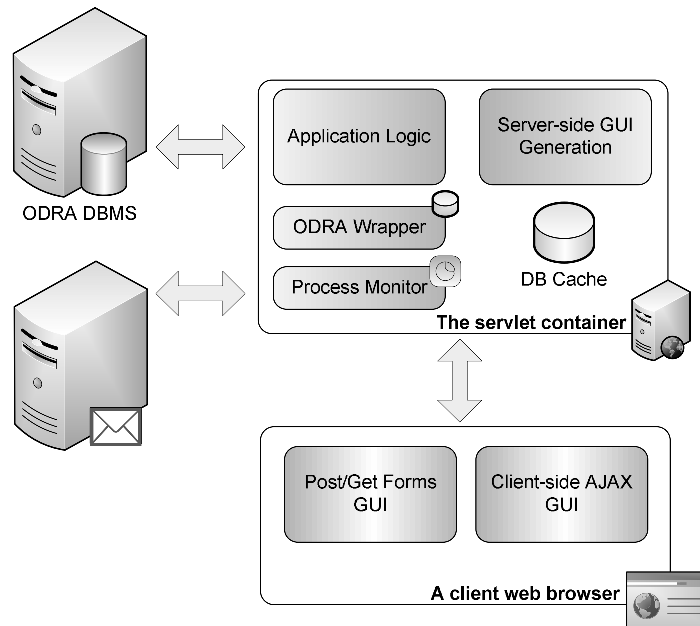

Fig.2. Overall architecture of the prototype

The main part of the system resides on the Tomcat 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.
- 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.3) or SBQL code editor with syntax highlighting (Fig.4). 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.

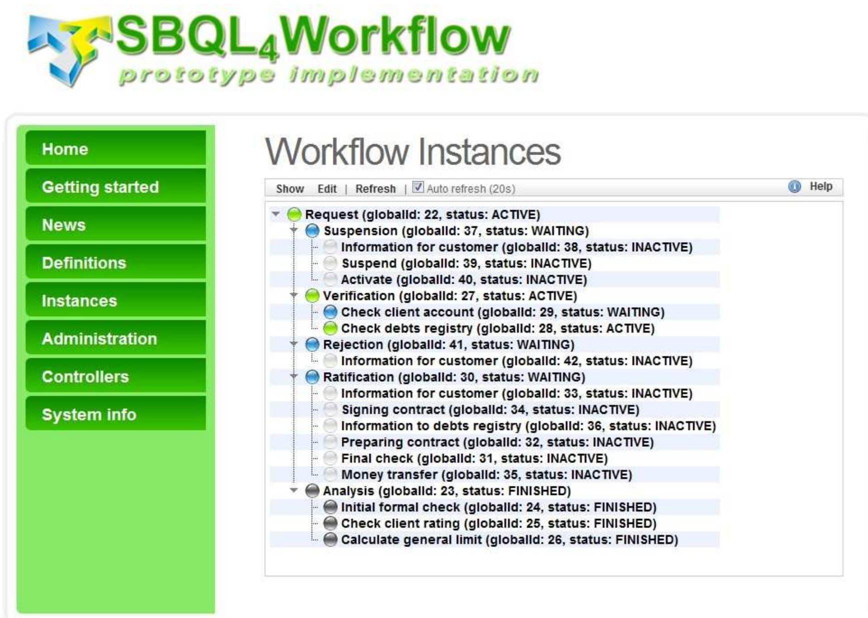

Fig.3. A workflow instances tree

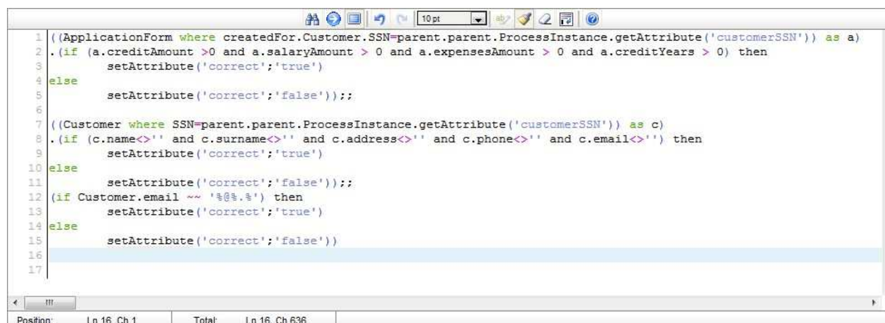

Fig.4. SBQL editor with a syntax highlighting

The last two remaining architecture's items from the Fig.2 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.5. 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-Children bidirectional pointer, combined with other 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 Process).

- Find my parent.
- Find a process with a given status.
- Find a process with a given name.

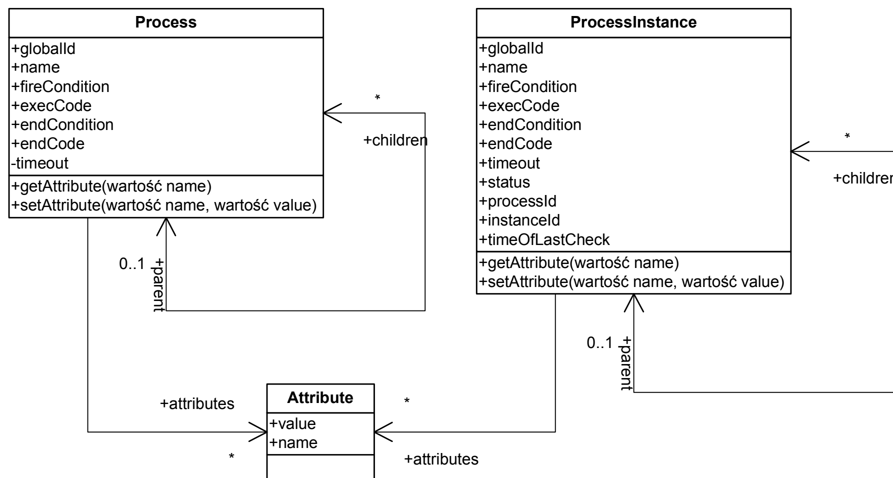

Fig.5. ODRA database schema

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 Process and ProcessInstance classes addressing atributes. Attributes can be easily used to control the flow (when the conditions are based on them) and allow communication between the Processes (as one Process can query other Process attributes and 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. The logic of the Monitor is described in the UML activity diagram in Fig.6.

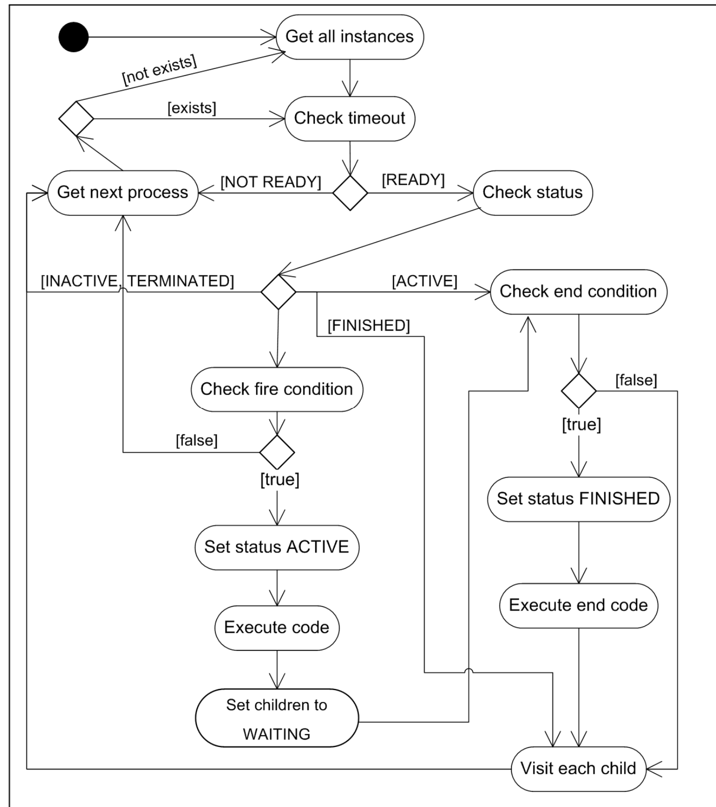

Fig.6. UML activity diagram representing the Workflow Monitor

4. Sample Declarative Workflow Definition

As illustration of our approach 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 of this object is in Fig.7. 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 which is entered into the database 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*).
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 and possibly archived in the database.
12. If at any stage the application is rejected the appropriate information is sent to the customer.

Let's 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 following condition have to be met:

```
exists(parent.ProcessInstance where getAttribute('state')='') and
exists(parent.children.ProcessInstance where name='Verification' and
status=ProcessStatus.FINISHED)
```

Firecondition of *Ratification*

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 in two ways:

1. All of its children are completed with the *finished* status
2. The application is rejected in the *Final check* sub-process, so the status of the *Final check* is *finished* and the attribute of the *Request* process has the value *rejected*.

```
(exists(parent.ProcessInstance where getAttribute('state')='rejected')
and exists (children.ProcessInstance where name='Final check' and
status=ProcessStatus.FINISHED)) or not exists(children.ProcessInstance
where status <> ProcessStatus.FINISHED)
```

Endcondition of *Ratification*

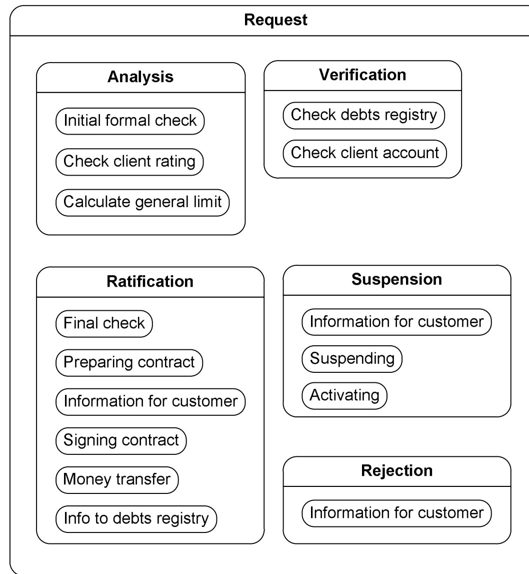

Fig.7. Structure of the *Request* process

When the Ratification is active, the fire condition of the child with the name *Final check* is satisfied:

```
exists(parent.ProcessInstance where status = ProcessStatus.ACTIVE)
```

Firecondition of *Final Check*

so the process activates and the following code is executed:

```
((ApplicationForm where createdFor.Customer.SSN =
parent.parent.ProcessInstance.getAttribute('customerSSN')) as a).(if
(a.salaryAmount - a.expensesAmount) > (a.creditAmount / (a.creditYears
* 12)) then
(parent.parent.ProcessInstance.setAttribute('state';'accepted')) else
(parent.parent.ProcessInstance.setAttribute('state';'rejected')))
```

Execution code of *Final Check*

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 corresponding fire condition is as follows:

```
exists(parent.children.ProcessInstance where name='Final check' and
status=ProcessStatus.FINISHED) and exists(parent.parent.ProcessInstance
where getAttribute('state')='accepted')
```

Firecondition of *Preparing Contract*

The purpose of this process is to create a new *Contract* object assigned to an application form filled by the customer. Below we present the execution code.

```
create Contract(now() as startDate, ref (ApplicationForm where
createdFor.Customer.SSN =
parent.parent.ProcessInstance.getAttribute('customerSSN')) as
attachment)
```

Execution code of *Preparing Contract*

It creates a new *Contract* object 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, as can be seen below:

```
exists(Contract where
attachment.ApplicationForm.createdFor.Customer.SSN =
parent.parent.ProcessInstance.getAttribute('customerSSN'))
```

Endcondition of *Preparing Contract*

Finishing of *Preparing contract* process activates *Information for customer*, with the following fire condition:

```
exists(parent.children.ProcessInstance where name = 'Preparing
contract' and status = ProcessStatus.FINISHED)
```

Firecondition of *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.

```
setAttribute('mailSent';(Customer where
SSN=parent.parent.ProcessInstance.getAttribute('customerSSN')).(sendMai
l(email;'Dear '+name+' '+surname+', We would like to inform that your
contract is prepared and waiting to sign up in our office.')));;
if (getAttribute('mailSent')='0') then (status :=
ProcessStatus.WAITING)
```

Execution code of *Information for customer*

When the e-mail is sent with no errors, 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. The fire and end conditions can look as presented below:

```
exists(parent.children.ProcessInstance where name='Information for
customer' and status=ProcessStatus.FINISHED)
```

Firecondition of *Signing contract*

```
getAttribute('contractSigned')='true'
```

Endcondition of *Signing contract*

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.

```
exists(parent.children.ProcessInstance where name='Signing contract'
and status=ProcessStatus.FINISHED)
```

Firecondition of *Money transfer*

The execution code for this process updates the *amount* attribute from the customer's *Account* object with the value of the *creditAmount* attribute from the specific *ApplicationForm* object. The process ends immediately after completing this action (no endcondition).

```
((Customer where SSN =
parent.parent.ProcessInstance.getAttribute('customerSSN')) as
c).((Account where owner.Customer.SSN=c.SSN).amount := (ApplicationForm
where createdFor.Customer.SSN=c.SSN).creditAmount)
```

Execution code of *Money transfer*

The last action in the ratification procedure is sending an info about a customer to a debts registry. We skip its obvious definition. 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.

5. Conclusion

We have presented the idea of an object-oriented declarative workflow management system that is especially prepared to achieve an important goal: the possibility of changing process instances during their run. We have discussed consequences of such a requirement and have argued that such a revolutionary feature cannot be achieved on the ground of traditional approaches to workflows based on specification of control flow graphs. Our idea allows to achieve next important features, such as mass parallelism of processes and flexible resource management. The idea is supported by the working prototype that shows its feasibility. The prototype is implemented on the basis of ODRA, an object-oriented distributed DBMS, and SBQL, a query and programming language designed and implemented for ODRA. In the paper we present a comprehensive example showing how a declarative workflow can be defined and how it can be dynamically changed.

The prototype is still under development, but we are applying for next grants that will allow us to turn it into a commercial (open source) product.

6. References

1. T.Andrews, F.Curbera, H.Dholakia, Y.Goland, J.Klein, F.Leymann, K.Liu, D.Roller,D.Smith, S.Thatte, I.Trickovic, S.Weerawarana. *Business Process Execution Language for Web Services, Version 1.1.* OASIS, 2003.
2. IBM developer works: Business Process Execution Language for Web Services, ver. 1.1, May 2003.
3. OMG. Business Process Modeling Notation (BPMN) specification. Final Adopted Specification. Technical Report, 2006
4. WfMC, WorkFlow process definition interface – XML Process Definition Language. WfMC-TC-1025 (Draft 0.03a); May, 22, 2001
5. 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
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. M.Momotko, K.Subieta. *Business Process Query Language - a way to make workflow processes more flexible*. Proc. 8th ADBIS'04, 2004, Budapest, Hungary, Springer LNCS 3255, pp.306-321, 2004
10. 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
11. 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
12. 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
13. M.Dąbrowski, M.Drabik, P.Habela, K.Subieta. *Object-Oriented Declarative Workflow Management System*. Editors of the Institute of Computer Science, Polish Academy of Sciences, ISBN 978-83-922508-3-8, 2009, 176 pages.
14. ODRA (Object Database for Rapid Application development): Description and programmer manual. http://www.sbql.pl/various/ODRA/ODRA_manual.html, 2008
15. K.Subieta. *Theory and construction of object query languages*. Editors of the Polish-Japanese Institute of Information Technology, 2005, 522 pages (in Polish)
16. K.Subieta. Stack-Based Architecture (SBA) and Stack-Based Query Language (SBQL). http://www.sbql.pl/, 2008
17. K.Subieta.:*Stack-based Query Language*. Encyclopedia of Database Systems 2009. Springer US 2009, ISBN 978-0-387-35544-3,978-0-387-39940-9, pp.2771-2772

18. K.Stencel. *Semi-strong Type Checking in Database Programming Languages*. Editors of the Polish-Japanese Institute of Information Technology, 2006, 207 pages (in Polish)