Prototype of Object-Oriented Declarative Workflows

Marcin Dąbrowski[1], Michał Drabik[1], Mariusz Trzaska[1], and 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. While in the traditional workflow processes the control flow is determined statically within process definitions, in declarative workflow processes the control flow is dynamic and implicit, determined by conditions that occur in the workflow data and the service environment. The environment consists of *active objects*, which play a double role. On the one hand, they 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 executable parts and represent workflow processes or tasks. The approach is motivated by features that are desirable in complex and less regular business processes: (1) the possibility of dynamic changes of process instances during their run, (2) mass parallelism of process instances and their components and (3) shifting the availability of resources that workflows deal with on the primary plan as a mean for triggering instances of process tasks. The paper presents the prototype of an object-oriented declarative workflows on a comprehensive example with roots in a real business case.

Keywords: workflow, object-oriented, declarative, query language, active object, dynamic workflow change, ODRA (Object Database for Rapid Application development), SBQL (Stack-Based Query Language).

1 Introduction

The workflow technology is a well developed domain with many commercial successes, which include such standards as BPEL [2], BPMN [8] and XPDL [13]. Nevertheless, there are still problems that undermine applications of workflow management systems in important business domains; in particular:

- Dynamic changes in workflow instances during their run. Dynamic workflow changes are the subject of many research papers, e.g. [1, 4, 5, 9, 10]. Although valuable results are achieved the problem in general still remains unsolved.
- Parallel execution of tasks within workflow processes. Currently, the parallelism is achieved by explicitly programmed splits and joins (AND, OR, XOR). In many cases such a parallelism is insufficient, for instance, when a process is to be split into many subprocesses, but their number is unknown in advance.
- Aborting a process or some of its parts. Currently, such situations are handled manually, with possibilities of inconsistencies and non-optimal human action.

- Resource management. In currently developed systems the control flow (a la Petri net) is on the primary plan. Resources necessary for execution (people, money, etc.) are on the secondary plan. But just availability, unavailability, planning and anticipating required resources are the main factors that should determine the process control flow.
- Tracking and monitoring. These activities should concerns the entire workflow environment and all running process instances, including databases that support workflows, the state of resources, anticipation of availability of resources, etc. For this reason the core for tracking and monitoring should be a query language (such as BPQL [6]) with the full algorithmic power rather than predefined tools.
- Parallel execution of workflow instances and their parts on many (hundreds) servers.
- Transaction processing. Classical implementations based on ACID properties and 2PC/3PC protocols are insufficient for workflows. Interactive business processes cannot be reversed, hence the attitude to transaction aborting should be changed.

In our last project we have investigated a new workflow paradigm that has the potential to overcome the above difficulties. We assumed that workflow instances can be changed during their run, hence they should possess a double nature. On the one side they are to be executable processes. On the other side, they should be considered database structures that are described by some conceptual schema and can be queries and manipulated as usual (nested) database objects.

The second assumption was inherent parallelism of all workflow processes and their parts. We avoid explicit splits and joins. Instead, we assume synchronization of parallel processes by special constructs of a query language. In this way our workflow instances remind PERT (Program Evaluation and Review Technique) networks rather than Petri nets. PERT naturally describes dependencies between tasks within non-computerized human activities and can be formalized using the object-oriented approach. Such a workflow system we describe as "declarative", because the control flow is not determined explicitly, but through declarative queries. Sequences of tasks can be supported by tasks' states and conditions on the states.

The third assumption is shifting the resource management on the primary plan. Resources (available, planned, anticipated) are reflected in the database. The control flow of process instances can be determined by conditions addressing resources.

In this way we came to the idea of *active objects*, which have the mentioned above double nature. Active objects are persistent data structures that are described by a database schema and can be queried and manipulated according to the syntax and semantics of a query language (in this role SBQL [12]). On the other hand, active objects possess active (executable) parts. We distinguish four kinds of such active parts: *firecondition*, *execution code*, *endcondition* and *endcode* (in this role SBQL too). An active object waits for execution until the time when its firecondition becomes true. After that, the object's execution code is executed, and all its active sub-objects are put into the *waiting-for-execution* state (and perhaps executed if their fireconditions become true). Execution of the execution code of a given active object is terminated when either all the actions are completed (including active sub-objects) or its endcondition becomes true. After fulfillment of an endcondition some actions might be necessary (e.g. aborting transactions), thus an optional endcode. Each active object is an independent unit that can be manipulated by SBQL functionalities (updated, deleted, etc.). Active objects can be nested. In this way they can represent workflow processes, their tasks, subtasks, etc. Active parts can also be updated; their parsing, type checking, optimization and compilation are performed on-the-fly. Bindings are mostly dynamic.

The paper [9] presents a 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 we do not determine explicitly the process control flow graph. It is on the secondary plan, determined dynamically and implicitly by fireconditions and endconditions. In majority of cases the control flow graph can be different depending on a runtime state of the workflow, database and computer environment. 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 [9], but it is not easy to see how such a feature can be achieved within the proposed formal workflow model. In our case the feature is an inherent property of the idea.

A declarative workflow deals with a database schema that describes executable data structures representing the processes, thus by definition enabling all updating operations that are provided within the assumed programming language SBQL. To restrict undesirable changes that may violate the consistency of processes we can use the semi-strong typing system that is implemented for SBQL. This of course may not be enough for more sophisticated situations. For this goal we plan to implement facilities that are well-know from relational systems, such as user rights, integrity constraints, business rules and triggers.

In [3] we describe in detail the concept of active object and related issues. To check the concept we have implemented three different prototypes. This paper is the first description of the third prototype [11], most advanced and with no previous tradeoffs concerning the new idea and current workflow technologies. The prototype is still a proof-of-a-concept rather a usable tool. More research and financial support is necessary to turn it into a product.

The rest of the paper is organized as follows. Section 2 presents basic assumptions and the architecture of the prototype. Section 3 presents how dynamic instance modifications can be performed. The presentation is based on a real example of a workflow that was taken from the experience in developing a bank system supporting credit processes. Section 4 concludes.

2 Prototype of Object-Oriented Declarative Workflow

The prototype [11] is built upon the ODRA system [7] and a Web-based application for manipulating prototype functionalities. The Web part uses the Groovy and Grails technologies. A workflow server part is written in Java.The prototype can be tested using a Web application called SBQL4Workflow. It allows for all administrative tasks like creating process definitions, manipulating them, instating processes, freezing parts of a running workflow and more. A GUI generation module is based on the core Grails framework technology called GSP (Groovy Server Pages). It is similar to JSP (Java Server Pages). A client side is equipped with advanced AJAX controls to allow dynamic loading of a process tree and manipulating workflow objects minimizing the need to reload web pages. The SBQL code editor with syntax highlighting that is included into GUI makes the work with workflows much easier.

The ProcessMonitor is a Java application that can be run as a separate thread on a separate machine. It periodically checks (basing on timeouts) each ProcessInstance. Then, according to the values retrieved from condition codes, the ProcessMonitor executes the execution code of the process.

The prototype is build using the standard three layer approach. A middle layer consists of the Application Logic and ODRA Wrapper. The corresponding API allows for work with workflow objects. It is used not only by GUI and the ProcessMonitor but can be used by any Java program, so writing a different client application is possible. The ODRA Wrapper is a wrapper between the JOBC (Java Object Base Connectivity) library that is used to access the ODRA DBMS through queries and Java business objects used by the Application Logic. All workflow data are stored on the ODRA DBMS.

The process objects represent structures created by the workflow programmer. Once a process is initiated, all data, including the data of sub-processes, are 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.

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

- Find a child that has a certain name and status.

  children **where** *name* = 'foo' **and** *status* = *processStatus.FINISHED*
- Check if all my children have the status ‘Finished’.

  exists(*children* **where** *status* = *ProcessStatus.FINISHED*)
- Find my “brother” (using *parent.children*).
- Find all my “nephews” (using *parent.children.children*).

To allow processes to store ad-hoc additional data we have provided the Attribute class with a set of methods in the Process and ProcessInstance classes addressing attributes. The attributes can be easily used to control the flow and allow communication between the processes (as one Process can query another Process attributes and change their values).

getAttribute('contractSigned')='true'

The code example presents the access to the attribute named ‘contractSigned’

setAttribute('mailSent'; *sendMail*('foo@bar.com'; 'Mail content'))

The code sends an email and stores the result (success or failure) as a process attribute.

3 Dynamic Instance Modifications

After creating a process instance for any business reason it can be the subject of changes without changing the corresponding process definition. Changes can be performed after launching an instance. Changes can also concern process definitions. Our prototype has the following options concerning changes within workflows:

- Editing and modifying process definitions;
- Instantiating process instances according to the definitions;
- Editing and modifying a process instance by editing its core attributes such as name, fire condition, end condition, execution code, end code, etc.;
- Running any SBQL program (having updates, inserts, deletions, etc.) in order to manipulate the entire workflow environment, including nested active objects representing processes, a resource database and any other persistent or volatile objects that are available within the environment. The programs include SBQL queries as expressions.

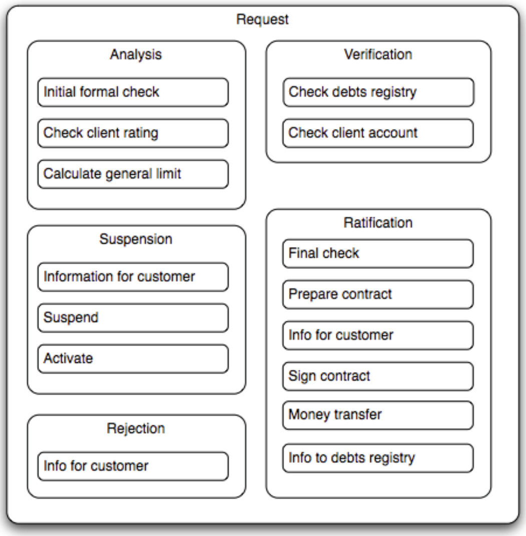

Fig. 1. ”Bank credit” workflow instance initial structure

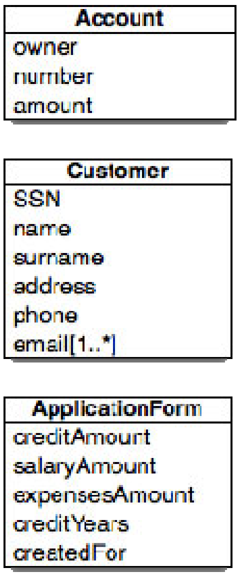

Fig. 2. Additional resources objects

Changing a process instance may require further changes of other instances to ensure consistency of the corresponding business process. Our prototype offers much flexibility in controlling process instances without altering other instances, mainly by preparing more generic fireconditions and endcondition that are insensitive to some changes of active objects. For instance, an endcondition can test completion of all corresponding sub-processes with the use of a universal quantifier. In many cases, however, altering a process instance may require some actions on other instances. These actions can be nested within a transaction.

To demonstrate the possibility of dynamic instance modification we have created a comprehensive example of real business processes concerning issuing and granting bank credits for customers. The structure (schema) of the process presented in Fig.1. All presented SBQL codes are tested on the prototype.

Example 1. *It demonstrates how to insert a new process into a workflow instance structure, without the need of changing the already working process instances details.* Letters in brackets correspond to status of a process instance: FINISHED, ACTIVE, WAITING.

Assume a bank credit process in progress. At the end of it the money that the customer has requested is transferred to his/her account. However after the transfer the customer has decided to change the target account. In this situation we can correct the working workflow instance by inserting an additional subprocesses that will do the requested operation. To achieve the goal we have to find a workflow instance that should be modified and create a new process called „*New account money transfer*" in it. It will have two attributes to store the value of an old and new account number, named respectively „*oldAccountNr*", "*newAccountNr*".

After inserting the new process its status is set to „*Waiting*". The next step is to set the proper firecondition. It should check whether exists the process "Money transfer" with the status equal to "FINISHED" at the same hierarchy level. It can be found within the children of "Ratification" process (the parent of the "*New account money transfer*"). The purpose of the newly inserted process is to withdraw the money from the old account and transfer it into the new one. The sequence of actions that needs to be performed is shown on the activity diagram below:

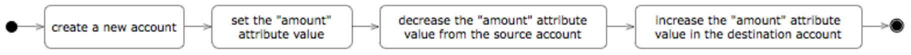
create a new account
set the "amount" attribute value
decrease the "amount" attribute value from the source account
increase the "amount" attribute value in the destination account

Fig. 3. Set of actions performed by "*New account money transfer*" execution code

The execution code creates a new account object with the number delivered from the „*newAccountNr*" attribute of this process instance. Now we should find out the information about the amount of money that should be transferred. This information is a part of the „*ApplicationForm*" object which is available, so the task will be to find the application form assigned to the current customer and obtain the „*creditAmount*" value. To make this value available for further processing it will be saved as a value of a newly created attribute called „*amount*". Now it is possible to withdraw the money from the old account. To do that the right Account object should be found (the account number is the value of the process instance „*oldAccountNr*" attribute), and the value of its „*amount*" attribute should be decreased by a value of this process instance „*amount*" attribute. Then the new Account object should be found (the account number is the value of the process instance „*newAccountNr*" attribute) and its "amount" attribute should be increased by the value of the process instance "*amount*" attribute.

Before making any changes to a working workflow instance it should be suspended so that the state before and after applying the change is consistent. Knowing the current state of all process instances we can assume that a newly created process should start when the „*Money transfer"* is finished, and should end when the transfer operation between accounts is complete. In this case the insertion of a new process instance doesn't influence any other process, the construction of „*Ratification"* end condition ensures that it will not finish before every of its child finishes.

Example 2. *This example demonstrates the possibility of modification of a running workflow instance structure in order to meet new requirements. It shows how the proper written execution code can modify behavior of the workflow instance and how the workflow administrator can influence the behavior.*

The customer has decided to increase the credit amount just before signing the contract. In that case there is no need to restart the whole workflow instance, but only some of the processes.

The activities that the workflow administrator have to perform are the following:

1. Suspend a chosen workflow instance.
2. Add new process instance "*Increase credit*" (as a child of "*Ratification*").
3. Delete process instances that are no longer required ("*Verification*" - child of "*Request*", and "*Initial Formal Check*" - child of "*Analysis*").
4. Change the conditions of other involved process instances to conform to the new structure.
5. Resume workflow instance.

The purpose of the newly created "*Increase credit*" process instance is to change the "*creditAmount*" attribute of the "*ApplicationForm*" object associated with the customer. Activity diagram shown below gives an overview of the actions performed by execution code:

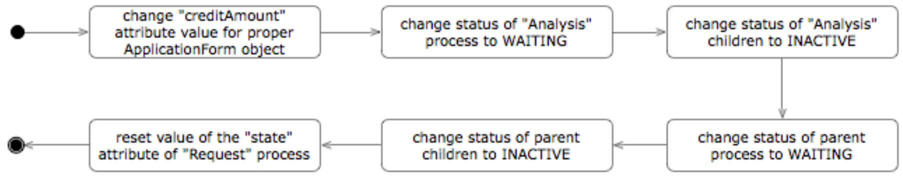
change "creditAmount" attribute value for proper ApplicationForm object
change status of "Analysis" process to WAITING
change status of "Analysis" children to INACTIVE
change status of parent process to WAITING
change status of parent children to INACTIVE
reset value of the "state" attribute of "Request" process

Fig. 4. Set of actions performed by "*New account money transfer*" execution code

It also resets the "Analysis" and "Ratification" process instances in order to perform their tasks once more. It is done by changing their status to "*Waiting*" so the process monitor will include them when checking the candidates to activate. The children of this processes should also be included with this difference that their status will be changed to "*Inactive"*.

Apart from process instances there are also attributes that values should be set to the previous state. This concerns the "*state*" attribute of the "*Request*" process instance, which holds the information about the current status of the application form and needs to be reset.

When the "*Increase credit*" will perform the given task it's no longer needed in the system and to ensure that it will act only once, we can create such an end code that will delete it.

The next step is to get rid of unnecessary process instances such as "*Initial formal check*" and "*Verification*", because there is no need to repeat them when only the amount of the credit is changed. After that the conditions of some process instances have to be adjusted. The "*Check client rating*" will start as soon as "*Analysis*" is active instead of start after the "*Initial formal check*" finishes.

All of the statements that concern "*Initial formal check*" should also be removed from the "*Analysis*" end condition.

The "*Ratification*" will no longer start after the "*Verification*" but as soon as the "*Analysis*" finishes.

Now the workflow instance is ready to properly handle the updated application form and perform suitable tasks in order to complete the request.

Example 3. *It demonstrates how to apply a modification that affects several process instances of the same kind.*

The modifications are to be applied to the „*bankLimit*" attribute of the „*Calculate general limit*" process to 700000. Changing the process definition is straightforward through the GUI tool. However, changing manually all of the working instances in this way is awkward and can be error prone. For this reason we create an SBQL statement which will access the workflow environment and will do the necessary modifications. The statement finds all of the instances of the "*Calculate general limit*" subprocess, which has an "*Inactive*" or "*Waiting*" status, and then updates the value of the "*bankLimit*" attribute to the new value.

An SBQL statement which updates the "Calculate general limit" "bankLimit" attribute in the proper workflow instances:

(*ProcessInstance* **where** *name* = 'Calculate general limit' **and** (*sta-tus*=*ProcessStatus.INACTIVE* **or** *sta-tus*=*ProcessStatus.WAITING*)).(*setAttribute*('bankLimit';'700000'))

Example 4. *It demonstrates how the working process instance can dynamically create new process instances.*

The commonly considered case in a process definition with parallel subprocesses is that a process instance is to be split into a fixed number of subprocesses. In many business situations the case leads to severe limitations, because the number of the subprocesses is known only during the execution of the instance. In such a case we should provide an option to create new subprocess instances dynamically, within the execution code of the process instance. To show this possibility we consider example where there is a need to send an e-mail with some information to the customer. During filling a request form a customer can provide some alternative e-mail addresses and we want to ensure that our e-mail will be delivered to all of them. The process responsible for the contact with the customer is "*Information for customer*" so we will modify it to provide required functionality. During the execution of this process instance it will create as many children process instances sending e-mails as required.

The execution code of the "*Information for customer*" creates a process instance for each of an e-mail address of a current customer:

(*self* **as** *p*).(((*Customer* **where**
SSN=*parent.parent.ProcessInstance.getAttribute*('customerSSN')) . *email* **as** *e*). (**create**
ProcessInstance(...)))

Then we can populate the execution code for the newly created process instances in such way that it will send mail for one given address. If an e-mail was sent successfully it will create an attribute "*mailSent*" with value 1 to hold the information which will be used later to decide if the process should end or restart. Below there is a part of the execution code of a dynamically created process instance that sends e-mail to a given address and sets an attribute value depending on the result:

setAttribute('mailSent';*sendMail*('x@x.x';'Dear Customer, your application is to be corrected.'));

Part of the execution code of a dynamically created process instance that restarts it when sending mail has failed:

if(*getAttribute*('mailSent')=0) **then** (*status* := *ProcessStatus.WAITING*);

4 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. In the paper we present comprehensive examples showing how a declarative workflow can be defined and how it can be dynamically changed. The examples have shown the feasibility of the idea of declarative workflows for real business cases. The prototype is still under development.

References

1. van der Aalst, W.M.P.: Generic workflow models: How to handle dynamic change and capture management information? In: Proc. 4th Intl. Conf. on Cooperative Information Systems (CoopIS 1999), Los Alamitos, CA (1999)
2. Andrews, T., et al.: Business Process Execution Language for Web Services, Version 1.1. OASIS (2003)
3. Dąbrowski, M., Drabik, M., Trzaska, M., Subieta, K.: Dynamic Changes of Workflow Processes (September 2010) submitted to publication
4. C.A.Ellis, C.A., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Proc. ACM Conf. on Organisational Computing Systems, COOCS 1995 (1995)

5. Ellis, C.A., Keddara, K., Wainer, J.: Modelling workflow dynamic changes using time hybrid flow. In: Workflow Management: Net based Concepts, Models, Techniques and Tools (WFM 1998), Computing Science Reports, vol. 98(7), Eindhoven University of Technology (1998)
6. Momotko, M., Subieta, K.: Process query language: A way to make workflow processes more flexible. In: Benczúr, A.A., Demetrovics, J., Gottlob, G. (eds.) ADBIS 2004. LNCS, vol. 3255, pp. 306–321. Springer, Heidelberg (2004)
7. ODRA (Object Database for Rapid Application development): Description and programmer manual (2008), `http://www.sbql.pl/various/ODRA/ODRA_manual.html`
8. OMG. Business Process Modeling Notation (BPMN) specification. Final Adopted Specification. Technical Report (2006)
9. Reichert, M., Dadam, P.: ADEPTflex: Supporting dynamic changes of workflow without loosing control. Journal of Intelligent Information Systems 10(2), 93–129 (1998)
10. Sadiq, S., Orlowska, M.E.: Architectural considerations in systems supporting dynamic workflow modification. In: Jarke, M., Oberweis, A. (eds.) CAiSE 1999. LNCS, vol. 1626, Springer, Heidelberg (1999)
11. SBQL4Workflow Prototype Implementation (May 2010), `http://tomcat.pjwstk.edu.pl:8080/ProjectWorkflow/newsitem/list`
12. Subieta, K.: Stack-Based Architecture (SBA) and Stack-Based Query Language, SBQL (2008), `http://www.sbql.pl/`
13. WfMC, WorkFlow process definition interface – XML Process Definition Language. WfMC TC 1025 (Draft 0.03a), May 22 (2001)