Mariusz Momotko

Tools for Monitoring Workflow Processes to Support Dynamic Workflow Changes

Ph.D. Thesis

Submitted to the Scientific Council of the Institute of Computer Science,
Polish Academy of Sciences

Advisor:
Prof. dr hab. Kazimierz Subieta

Warsaw, Poland, January 2005

Abstract

During the last decade workflow management has had a successful career. Workflow management technology has become a critical component of IT applications, as have databases, transaction processing or user interfaces. Despite many advantages resulted from application of workflow management systems, significant limitations were also observed. One of the major restrictions was the assumption that business processes do not change very often during their execution. Such an assumption does not hold for most of real, less rigid business processes which need to adapt to dynamic changes in the workflow environment (i.e. data, resources, applications) as well as the workflow itself.

This thesis aims at proposing a new comprehensive approach to increase the adaptability of workflow management systems by making the definition of workflow processes more flexible. The proposed approach is based on the exploitation of workflow monitoring tools within a process definition to adapt dynamic workflow changes.

In order to achieve the above goal, first we proposed a generalised workflow process metamodel. This metamodel includes data on the workflow environment, process definition and process enactment. On the basis of the metamodel we defined a powerful and flexible object-oriented query language to query it, specifically, the Business Process Query Language (BPQL). This query language was derived from Stack-Based Query Language and its fundamental concepts have been taken from the stack based approach to query languages. Having BPQL, we designed a set of 32 workflow monitoring functions concerning process execution history, selection of the best candidates for workflow participants or information about the current values of quality of services. In the next step we showed how BPQL and the designed monitoring functions may be applied in a workflow process definition and proposed how to integrate smoothly BPQL with a standard process definition language proposed by Workflow Management Coalition, namely XML Process Definition Language (XPDL). Afterwards, a prototype version of BPQL and workflow monitoring functions has been developed within the Intelligent Content Management System, an IST EU research project, and has became an intrinsic part of the OfficeObjects®WorkFlow system. Finally, the usefulness of the overall approach and the developed tools has been verified in their implementation in the European Exchange of Documents-Poland system.

Narzędzia do Monitorowania Procesów Pracy Wspierające Dynamiczne Zmiany Procesów

Streszczenie

W ostatnim dziesięcioleciu zarządzanie procesami pracy robi zawrotną karierę. Technologia związana z zarządzaniem procesami pracy staje się kluczowym elementem systemów informatycznych, tak jak: bazy danych, przetwarzanie transakcyjne czy interfejs użytkownika. Pomimo wielu korzyści płynących z zastosowania systemów do zarządzania procesami pracy widoczne są też ich znaczące ograniczenia. Jednym z głównych jest założenie, że procesy biznesowe nie zmieniają się w trakcie ich wykonania. Założenie takie jest nieprawdziwe dla większości rzeczywistych procesów wymagających adaptacji do dynamicznych zmian zarówno w samym przepływie pracy, jak i otoczeniu procesu (dane, zasoby, aplikacje).

Celem niniejszej pracy jest zaproponowanie kompleksowego podejścia do zapewnienia elastyczności procesów i adaptacji do dynamicznych zmian. Fundamentalną koncepcją jest wykorzystanie narzędzi (funkcji) do monitorowania procesów (operujących na danych z wykonania procesów) bezpośrednio w definicji procesu.

Po pierwsze, aby osiągnąć zadany cel, zaproponowano generyczny metamodel procesów zawierający dane o środowisku procesów (system informatyczny), ich definicji i wykonaniu. W oparciu o zaproponowany metamodel zdefiniowano ekspresywny i elastyczny język do odpytywania metamodelu, a mianowicie Business Process Query Language (BPQL). Język ten wywodzi się z języka Stack-Based Query Language a jego podstawowe koncepcje zostały zapożyczone z podejścia stosowego (stack-based approach) dostosowanego do języków zapytań. W oparciu o BPQL został zdefiniowany pakiet 32 funkcji do monitorowania procesów. Funkcje te wspierają operacje dotyczące historii wykonania procesu, wyboru najlepszych wykonawców czy też uzyskania informacji o aktualnych wartościach parametrów jakościowych. Następnie pokazano jak BPQL i wspomniane funkcje mogą być użyte w definicji procesu oraz jak w prosty sposób zintegrować BPQL ze standardowym językiem definicji procesów - XML Process Definition Language (XPDL). W końcowym etapie, wykonano prototypową implementację języka BPQL i wybranych funkcji do monitorowania procesów. Implementacja ta jest integralną częścią systemu OfficeObjects®WorkFlow i została wykorzystana w komercyjnym systemie Europejskiej Wymiany Dokumentów – Polska. System ten jest odpowiedzialny za obsługę korespondencji pomiędzy Radą Unii Europejskiej a Rzeczpospolitą Polską.

Acknowledgements

To Sylwia, my wife and Marcelina and Adrianna, my daughters who gave the sense of this work.

To Krystyna and Walenty, my parents who doubted if I would be able to do this work.

To professor Kazimierz Subieta, my supervisor, and dr Witold Staniszkis, the president of Rodan Systems S.A. who showed me how to pursue the goal of this work.

To many others, especially dr Bartosz Nowicki, Michał Gajewski, Wojciech Izdebski and Paweł Urbański.

Table of Contents

1 INTRODUCTION 1

1.1 MOTIVATION 1
1.2 STATE OF THE ART AND WORKFLOW STANDARDS 2
1.3 KEY WORK OBJECTIVES AND APPROACH 4
1.4 RELATED WORK 5
1.5 THESIS STRUCTURE 6

2 WORKFLOW PROCESS METAMODEL 7

2.1 IMPLEMENTATION BASE 8
2.2 WFM SYSTEMS AS PART OF IT SYSTEMS 9
2.3 DESCRIPTION OF THE METAMODEL 10
2.4 REQUIRED ELEMENTS OF IT SYSTEMS 15
2.5 PROCESS DEFINITION 18
2.6 PROCESS ENACTMENT 32
2.7 BASIC WORKFLOW TYPES 43
2.8 CLOSING REMARKS 47

3 BUSINESS PROCESS QUERY LANGUAGE FOUNDATIONS 49

3.1 REQUIREMENTS FOR BPQL 50
3.2 WORKFLOW MANAGEMENT 55
3.3 STANDARD QUERY LANGUAGES 56

4 BUSINESS PROCESS QUERY LANGUAGE FRAMEWORK 71

4.1 SYNTAX 71
4.2 SEMANTICS 74
4.3 PRAGMATICS – BY EXAMPLE 96

5 BUILT-IN WORKFLOW MONITORING FUNCTIONS 99

5.1 PROCESS FLOW FUNCTIONS 103
5.2 WORKFLOW PARTICIPANT ASSIGNMENT FUNCTIONS 107
5.3 INTRODUCTION TO QUALITY OF SERVICE FUNCTIONS 111
5.4 TIME FUNCTIONS 111
5.5 COST FUNCTIONS 115
5.6 RELIABILITY FUNCTIONS 116
5.7 APPLICATION EXAMPLES 117

6 BPQL APPLICATIONS 123

6.1 INTEGRATION OF BPQL INTO XPDL 123
6.2 THE INTEGRATED WFM + BPQL ARCHITECTURE 128
6.3 DETAILED BPQL ARCHITECTURE 132

6.4 IMPLEMENTATION IN OFFICEOBJECTS® 135
6.5 THE POLISH GOVERNMENT – EUROPEAN COMMISSION INTEROPERABILITY 136

7 CONCLUSIONS AND FUTURE WORK 143

REFERENCES 145

APPENDIX A: BPQL SYNTAX IN EBNF 151

APPENDIX B: DEFINITION OF THE BUILT-IN WORKFLOW MONITORING FUNCTIONS... 153

APPENDIX C: BPQL MANAGER INTERFACES 169

APPENDIX D: METAMODEL GLOSSARY 175

APPENDIX E: BPQL GLOSSARY 177

APPENDIX F: LIST OF FIGURES 178

APPENDIX G: LIST OF TABLES 179

List of Abbreviations

BPEL	*Business Process Execution Language*
BPM	*Business Process Management*
BPMG	*Business Process Management Group*
BPMI	*Business Process Management Initiative*
BPMN	*Business Process Modelling Notation*
BPQL	*Business Process Query Language*
CORBA	*Common Object Request Broker*
DBMS	*Data Base Management System*
EBNF	*Extended Backus-Naur Form*
EJB	*Enterprise Java Beans*
ENVS	*ENVironmental Stack*
ODMG	*Object Data Management Group*
OGSA	*Open Grid Service Architecture*
OMG	*Object Management Group*
OOQL	*Object-Oriented Query Language*
OQL	*Object Query Language*
QRES	*Query REsult Stack*
RDBMS	*Relational Data Base Management System*
RPN	*Reverse Polish Notation*
SBA	*Stack- Based Approach*
SBQL	*Stack- Based Query Language*
SOAP	*Simple Object Access Protocol*
SQL	*Structured Query Language*
UDDI	*Universal Description, Discovery and Integration*
UML	*Unified Modeling Language*
WARIA	*Workflow and Reengineering International Association*
WfM	*Workflow Management*
WfMC	*Workflow Management Coalition*
WfMS	*Workflow Management Systems*
WPAL	*Workflow Participant Assignment language*
WPDL	*Workflow Process Definition Language*
WSCI	*Web Service Choreography Interface*
WSDL	*Web Service Description Language*
XML	*eXtended Markup Language*
XPDL	*XML Process Definition Language*

"Everything flows, nothing stands still."
(In Greek) „Παντα ρει και ουδεν μενει"
Heraclitus of Ephesus, (535-475 BC)

1 Introduction

During the last decade workflow management (WfM) made a successful career. The WfM systems have been used for implementing various types of business processes. At least there are two reasons for such popularity of workflow management.

Firstly, workflow management technology is the confluence of management trends with computing and communication capabilities. Management trends include reinventing and revitalising of companies and corporations through business process reengineering and by increasing their focus on end-users' needs. Technical factors include significant progress in communication and distributed computing infrastructure technologies. Recently, the latter increasingly involves Web services, Grid and component technologies (WSDL, UDDI, SOAP, OGSA, Corba, EJB, etc.).

Secondly, workflow management introduces a new quality in computer technology by ability to monitor executed processes. The result of such monitoring is the extended knowledge about processes (e.g. quality of service factors, workflow participants, typical behaviours) and opportunities for their optimisation, thus more effective execution in the future. In contrast, in the traditional programming there is no such controlling mechanism available and such knowledge is neither collected nor used for the future executions.

Workflow management technology becomes a critical component of IT applications, just like database, transaction processing or user interface are. Usually, this technology is also perceived as a bloodstream of every modern organisation.

1.1 Motivation

Despite many advantages resulting from application of WfM systems, there were also observed significant limitations. One of the major restrictions was assumption that business processes do not change frequently during their execution. While such assumption may hold for majority of production processes, for less rigid processes, such as administrative ones, that is not true. Because of the nature of the latter processes, they need to adapt to **dynamic changes** ([Sadiq00], [Aalst99], [Weske99]) in workflow environment (i.e. resources, data and applications) as well as workflow itself (e.g. the current workload of participants). For example, the head of a department would like to select a person who will process a given claim. This selection should be done dynamically from

the list of all employees in this department. Another example is related to optimisation of the car repairing cost, when depending on time and the current cost of repairing either we make additional verification or not.

As it is reported in the last surveys, **workflow flexibility** is regarded as one of the key requirements for modern WfM systems. For example, as it was stated in Vector's survey conducted in more than 200 American companies "mid-sized manufacturers are still challenged with regard to enabling technologies that give them more business process flexibility" [Vector00]. Also a recent Ernst & Young's survey showed than „more than 40% of executives up to CEO level are looking into flexibility as a key strategic focus (telecom equipment manufacturers)" [Buttler02]. Moreover, flexible workflow processes are also considered together with shared knowledge and continuous process improvement as a crucial element of so-called knowledge era [Moore00].

An approach to increase processes adaptability is to make their definition more flexible. In this context flexible means that it is possible to express within a workflow process definition complex dynamic requirements that depend on process execution history as well as current organisational and application data. An alternative is manual control of processes and their resources at run time and this may be the only way for complex and unpredictable cases. However for many quite complex requirements, such as workflow participant assignments and transition conditions, the former approach seems to be the closest to the real business process behaviour.

Unfortunately, so far there is no coherent, complete, and semantically correct approach how to make process definition more flexible in order to cope with dynamic changes. Existing approaches and solutions are either too restricted or do not provide coherent semantics. What is more these approaches and solutions are not readable enough for process designers who, in general, are non-IT professionals. Before we discuss our approach to cope with dynamic changes let us check what was done on the business process and workflow standards within the last decade.

1.2 State of the Art and Workflow Standards

In early 90s increasing number of workflow applications specified two new important requirements for WfM systems, namely integration with existing applications/services and co-operation among different WfM systems. The above requirements were the starting point for international effort to standardise workflow description and WfM systems. So far there are five most generally recognised standardisation bodies that published standards on workflows: WorkFlow Management Coalition (WfMC, 1993), Workflow and Reengineering International Association (WARIA, 1992), Object Management Group (OMG, 1998), Business Process Management Group (BPMG, 1999) and Business Process Management Initiative (BPMI, 1999).

During the last decade, the WfM Coalition proposed a workflow standards framework that includes more than ten standards and associated documents concerning WfM systems. In the first

place, the coalition defined a standard architecture of a WfM system [WfMC-TC-1003]. This architecture, known as the *reference model* defined a WfM system as a system that consists of six modules, that is Process Definition Tool, Workflow Engine, Client Applications, Invoked Applications, Administration and Monitoring tools and Workflow Engine Interoperability. On the basis of this model the WfM coalition specified interfaces between Workflow Engine and the rest of the above mentioned modules. These interfaces have been published as separate standards [WfMC-TC-1016-P, WfMC-TC-1009, WfMC-TC-1012, WfMC-TC-1015]. Together with the reference model, the coalition presented a workflow process model known as the **process metamodel** [WfMC-TC-1011]. This model specifies the basic entities of a workflow process such as activity, workflow participant, transition, etc. On the basic of the metamodel the coalition proposed a language to represent workflow process definition, namely Workflow Process Definition Language (WPDL). In WPDL a workflow process was described in a textual form according to the WPDL grammar. After popularisation of markup languages, WPDL has been adopted to eXtensible Markup Language (XML) representation and is known as XML Process Definition Language (XPDL) [WfMC-TC-1025].

On the market, there is another competitive language to define business processes: Business Process Modelling Language (BPML) from BPMI. In BPML a workflow is also represented using the XML notation. Basically, it offers similar functionality as XPDL does, however detailed specification is quite different (e.g. SPLIT and JOIN operations - cardinality). Some issues on similarities and differences between these two languages have been presented in [Shapiro02].

Since the interface available for workflow interoperability specified by the WfM coalition had been criticised for its complexity and inadequacy in a distributed environment, two other protocols have been suggested: Simple Workflow Access Protocol (SWAP) [Swenson98] and jFlow from OMG [jFlow98]. Both protocols use the HTTP protocol to exchange information between different WfM systems. In addition, the authors of jFlow suggest that the reference model should also be simplified in order to be appropriate for Web communication. As a response for such effort, the WfM coalition presented a new workflow interoperability protocol, namely Wf-XML Binding [WfMC-TC-1023]. This protocol includes ideas presented in the WfMC's MIME binding specification as well as those in SWAP and jFlow. In Wf-XML, HTTP protocol is used as a data transport mechanism for exchange of Wf-XML messages. The body of such a message is written in XML.

Recently, as an answer for the increasing need to integrate Web services into WfM systems, there have been proposed some extensions either to workflow definition languages or to web service description languages. At the end of July, 2002 IBM, BEA and Microsoft presented the first version of Business Process Execution Language for Web Services (BPEL4WS) [BPEL4WS]. This language is based on work presented in BPML and mainly extended of execution web services

as activities. Another example is Web Services Choreography Interface (WSCI) proposed in June, 2002 by Sun, BEA, Intalio and SAP [WSCI02]. At the end of July also the WfM Coalition extended XPDL adding execution of Web services.

1.3 Key Work Objectives and Approach

There are two main aims of this work. The first one is to define a language and a set of tools (or functions) to monitor workflow processes and to use them within a standard and widely know process definition language in order to make it flexible and able to cope with dynamic changes. The second one is to verify practically the usefulness of the proposed approach by implementing a prototype and then consequently applying it in a real application.

In order to achieve the above goals first we define an appropriate process metamodel, develop a language to query this model, then build workflow monitoring functions within this language and integrate this query language with a standard process definition language. Finally, we develop a prototype version of the language interpreter and monitoring functions and verify their usefulness within a real practical application.

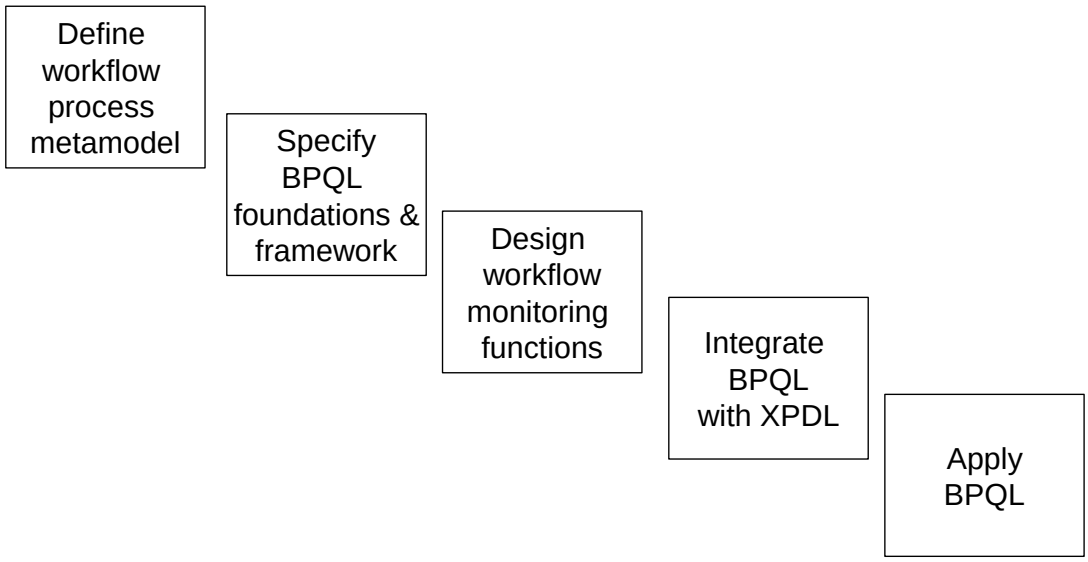

Figure 1.1 Required steps of the proposed approach

The process metamodel is generic and includes both process definition as well as process execution entities. It is an extension of the WfMC process definition metamodel. It also includes some process features proposed within the last workflow research such as advanced time management [Eder97, Eder99, Eder01] or flexible workflow participant assignment [Momotko02a].

The language to query the mentioned metamodel is called the Business Process Query Language (BPQL), which is an object oriented language. It is based on the Stack-Based Approach (SBA) [Subieta95a] and Stack-Based Query Language (SBQL) [Subieta95a, Subieta95b,

Subieta04]. From SBA and SBQL roots BPQL inherits coherent and complete semantics as well as clear and readable syntax.

The workflow monitoring functions are built on the top of BPQL. All together, there are 32 monitoring functions defined. They are divided into three groups: process flow functions, participant assignment functions and Quality of Service (QoS) functions.

BPQL is integrated within XPDL. The XPDL+BPQL extension has been build in such a way that it does not require to modify the original XPDL grammar.

The first prototype BPQL version together with selected monitoring functions have been implemented within the OfficeObjects®WorkFlow system as OfficeObjects®BPQL [Momotko04]. This work was done within the EU Intelligent Content Management System IST project (ICONS, FP5, see www.icons.rodan.pl). In the first quarter of 2004, the implemented prototype was applied in the European Exchange of Documents-Poland (EWD-P) system which is responsible for preparation of the official Polish position concerning adjustment of the newly associated states law to the EU procedures and regulations [Blizniuk05].

1.4 Related Work

There is at least one widely known process definition metamodel proposed by WfM Coalition. While this metamodel is well defined, there is no standard process execution metamodel provided neither by WfMC standards nor by other process management body (e.g. BPMI). In addition, process execution models provided by WfM systems seem to be tool oriented and mainly focusing on entities implemented within a given system. Recently, there is some effort in defining such generalised process execution model (e.g. [List03]). However, it does not include some process features proposed within the last workflow research such as advanced time management or flexible workflow participant assignment.

So far, there are a few other approaches to define a business process query language and use it in process definition. Firstly, in [Ceri03] the authors present a language to model web applications – WebML. This language provides functionality to define business processes and make them more flexible. To define a condition or a workflow participant it is possible to use an attribute whose value may be calculated by a complex program. Despite its flexibility (it is algorithmically complete) this approach seems to be less appropriate for non-programming process owners and to hide the algorithm to calculate this attribute inside the program code. In addition, to the best of our knowledge this language is not compliant with any of the existing well known standard process definition languages.

In [Momotko02a] the authors proposed a Workflow Participant Assignment Language (WPAL) – a functional language to define workflow participant assignment. This language is able to use workflow control data (e.g. the performer of a given activity, reference to the objects

represents the start activity). Despite its flexibility, it has similar problems as WebML. BPQL may be treated as a continuation and significant extension of WPAL.

Another interesting example of a query language to measure workflow performance has been presented in [Abate02]. The authors proposed, the Workflow Performance Query Language (WPQL)which is also a functional language and provides a set of basic operators to operate on selected process enactment entities (process instance, activity instance, and workitem). On the basis of these operators it is possible to define more advanced functions to calculate selected factors of workflow processes. Unfortunately, in the documents known by the thesis author there is little information on WPQL syntax and semantics. Also flexibility of WPQL to build new functions on the selected process enactment entities which, for real queries on workflow metamodel, may be not too restrictive.

Finally, there is some work on a business process query language carried out by BPMI (see www.bpmi.org). It is promised that this language will offer facilities for process execution and process deployment (process repository). However, after two years of this work, still there is no official, even a draft version available.

On the other hand there are many workflow management systems that provide some selected elements of business process query languages. For example in Staffware [Staffware99], there is a set of workflow functions (i.e. SW functions) that can be used to define conditions and workflow participant assignments. In Lotus Workflow [Lotus01], it is possible to call a lotus script function in order to define the above elements. OfficeObjects®WorkFlow in a previous version implemented WPAL. All these examples of query languages do not provide clear and coherent semantics and support a small sub-set of possible queries only. In addition, the problem with moving algorithms from process definition into application still remains.

1.5 Thesis Structure

The structure of the remaining chapters of this thesis is as follows. Chapter 2 describes the workflow metamodel. This metamodel defines process definition and process execution entities, associations among them, their attributes and behaviours. Chapter 3 presents the BPQL foundations focusing on requirements for the type of the query language and its influence on the workflow management frameworks. Chapter 4 defines BPQL that is its syntax, semantics and pragmatics. Chapter 5 includes specification of 32 workflow monitoring functions written in BPQL. The BPQL code of these functions is included in Appendix B. Chapter 6 proposes the way how BPQL may be integrated in XPDL, details the development issues connected with a prototype BPQL implementation in OfficeObjects®WorkFlow and describes an application of BPQL within the EWD-P system. Chapter 7 provides some conclusions and discusses possibility for future enhancements of BPQL.

2 Workflow Process Metamodel

The workflow process metamodel is a conceptual view (a schema) over a data structure that maps all the entities and factors taking part in the process definition and execution. In particular, the metamodel maps the structure of process definitions and the structure of executed processes (activities, transitions, states, assignments) thus treating processes as data structures with entities associated with other entities of the workflow environment. The metamodel may also take into account arbitrary business context that the processes address, for instance, workflow participants, organisational structure of an enterprise, various resources that are necessary to execute the processes (documents, manpower, time, cost, infrastructure, etc.), and so on. The metamodel can also take into account not only currently running processes, but also historical and suspended processes, as well as processes that will be executed in the future. Thus in general, the metamodel can take into account all past, present and future entities and factors that may directly or indirectly (via business processes) influence the run of any particular process.

Moreover, the workflow metamodel can also include special elements that appear due to attempts to control or datamine the entire population of processes. Examples of such elements concerns preclaiming some resources by particular processes, reports on used resources, reports on used documents, reports on workflow participant assignments, statistics, datamining results, etc. Such elements may be required to define sufficiently flexible workflow monitoring functions.

Examples of queries addressing such a metamodel are the following:

- *Get all the documents that were processed by any process in the last month.*
- *For each employee get the total number of hours that was spend by him/her in the last week for serving workflow processes.*
- *Get all workflow processes that have processed on the document D78-90;*
- *For each document processed in the last week get the list of workflow participants that have read or modified it.*
- *Which workflow process is currently processing the document D78-90?*

On the other hand, taking into a workflow process metamodel all the above entities and factors may cause the model to be too complex for users/programmers, too difficult to maintain and making the necessity of extra human effort in during business processes. For these reasons a workflow metamodel is a tradeoff between expressiveness (ability to retrieve any information from the workflow and business environment) and simplicity. A good metamodel should take into account such entities and features that are easy to capture within a workflow environment, easy to maintain, are the subject of frequent dynamic changes during execution of processes and are

critical to the behaviour of processes. Careful analysis of all the objects and factors within the entire workflow and business processes environment is the basis to recognise which of them are most promising to be recorded as elements of the metamodel.

2.1 Implementation Base

After discovering entities and factors essential for the metamodel construction the next important question concerns representation of them as a data structure within the computer storage. In a typical scenario the metamodel can be developed in UML as a class diagram. Then, the diagram becomes directly the schema of metadata implementing the metamodel. For several reasons, however, this scenario is rarely practical. So far there is no DBMS which directly accept an UML diagram as a database schema. Moreover, the choice is usually constrained by available or assumed implementation environment. Currently we can consider the following options:

- Relational DBMS,
- Object-relational DBMS,
- XML-oriented or RDF-oriented repositories,
- Object-oriented DBMS.

Each option has pros and cons. An advantage of the first option is market authority and availability of relational systems. This option, however, implies severe warping the conceptual model that has been developed at the beginning. According to our experience, mapping an UML class diagram into a relational schema much increases the complexity (usually there are 2-3 times more relational tables than classes in the corresponding UML diagram. Moreover, some information will be lost, e.g. information on inheritance, some cardinalities and relationships. This causes that a relational schema, in comparison to an UML diagram, is much more difficult for users and much more error prone. Also queries in SQL will be complex, due to the bigger number of database entities and due to join predicates that have to associate the tables participating in a query.

Just on the reverse pole there are object-oriented DBMSs. So far their market authority is rather small. The reason is obvious: vendors of relational systems are interested in selling their current products rather than supporting a kind of revolution that is implied by object-oriented systems. In nearest years this probably will be changing, due to big conceptual advantages of object-oriented databases over their relational rivals. Object oriented databases minimize the gap between a conceptual schema (in UML) and a database schema.

Somewhere in the middle of the above two extremes are object-relational databases and XML and RDF repositories. Unfortunately, object-relational databases seem to fall short of advantages of object-orientedness. As a rule, the object-oriented features are artificially and inconsistently stuck to a major relational kernel, thus usability of them is low or none. David

Maier, in his famous interview concludes: „I don't see any end application users using the new object-relational features" [Winslet02]. Hence from for our purposes object-relational databases share advantages and disadvantages of pure relational databases. A similar case concerns XML and RDF repositories, which are not mature enough yet and still introduce conceptual limitations in comparison to UML class diagrams (no associations, no inheritance, etc.).

Therefore in this thesis we do not follow the conservative approach forced by the current market tendencies. We conclude that only pure object-oriented databases present the proper target platform for implementing a workflow metamodel. The above discussion we will continue during analysis of candidate query languages addressing the metamodel.

Summing up, in order to specify a business process query language, in the first stage an appropriate workflow process metamodel should be defined. This metamodel needs to represent two parts of 'workflow process puzzle' that is process definition as well as process execution. The former part is mainly used by workflow engines to execute workflow processes while the latter helps monitoring and analyzing workflow process execution. In addition, the model is represented on two levels of abstraction: conceptual level and database schema level. The metamodel (or its part) presented at the latter level is a more detailed description of that one presented at the conceptual level. Both represent classes and associations amongst them. The conceptual view presents the name of the classes, the name of associations and multiplicity of the classes which participate in associations. The database schema view presents the name of the classes, and their attributes (names and types), the multiplicity of classes participating in the associations and the name of the pointers (instead of roles).

2.2 WfM systems as part of IT systems

WfM systems may be considered as a part of IT systems which is responsible for their dynamics. WfM systems require from IT systems three elements: *data* that they are processing, *services* that are provided, and finally, registered *resources* that may execute the mentioned services operating on the data.

Part of data managed by IT systems is used by WfM systems to control execution of workflow processes (i.e. within flow conditions, and workflow participant assignment). The WfM systems have rights only to read these data, not to modify them. According to the WfMC terminology these data are called *workflow relevant data*. The workflow relevant data are accessed by WfM systems in read-only mode. Services provided by IT systems are used to support execution of the process activities. During execution of activities WfM systems simply call these services with appropriate parameters. The called services may return the result which is then mapped into appropriate data container attributes. Such a service is called an *application*. There are also resources - users or automatic agents that may perform some activities within workflow processes.

Resources may be classified along roles (sharing some characteristics), groups or organizational units. A resource that participates in process execution is called a *workflow participant*.

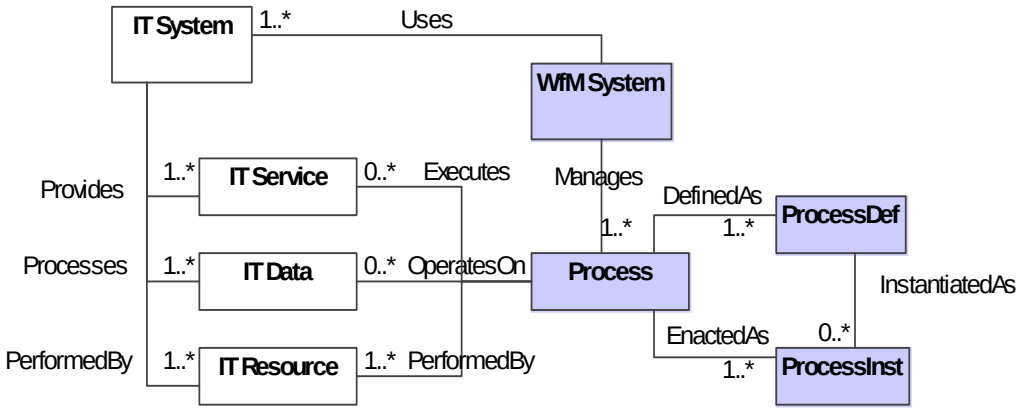

Figure 2.1 Basic relations between IT system and WfM system

In addition to the above elements WfM systems use *workflow control data*. These data are managed only by WfM systems and store workflow specific information such as number of active process instances, the current state, etc.

The above description of IT – WfM systems' dependencies is just an introduction.

2.3 Description of the Metamodel

The workflow process metamodel described in this chapter defines workflow entities, and relationships among them. It consists of two parts, namely the process definition metamodel and the process instance metamodel. The former metamodel defines the top-level classes contained within a process definition, and associations among them. It is used to design and implement a computerised representation of a business process. The latter model defines the top-level entities contained within process instantiation, and associations among them. This model is used to represent process execution which is being done according to the process definition model. The metamodel also shows how the individual process definition entities are instantiated during process execution.

On of the main entities of the metamodel is *process definition*. This entity represents basic information about the computerised representation of a business process. Since this representation may change during process operation, it is possible to have more than one version of the process definition for a given business process. For every process a set of *container attributes* is defined. These attributes may be considered as workflow system and environmental data. They may be used during process execution in the evaluation of conditional expressions such as transition condition or pre and post conditions. The set of container attributes (i.e. number, types, and names) strictly depends on individual process definitions. In addition, a process may have *parameters* . There are

two types of parameters: input and output (results) ones. The process parameters may be mapped into appropriate container attributes.

Process definition consists of activities (at least two). An *activity* defines a piece of work that forms one logical step within a process. There are three types of a process activity: atomic, routing and compound ones.

An *atomic activity* is a smallest logical step within the process that may not be further divided. For every atomic activity it is possible to define *who* will perform it, *how* it will be executed and *what* data will be processed. An atomic activity may be performed by one or more workflow participants. Basically, a workflow participant is a user (or their group), a role or an organizational unit. Also automatic agents can be treated as a kind of workflow participants. Specification of workflow participants that may perform a given activity is called *workflow participant assignment*.

An *application* specifies how activity will be performed. Such specification also includes a set of parameters that will be passed to the application. Since the mentioned application operates on data, also *object types* that will be processed (i.e. created, modified, read or deleted) by this activity have to be defined. Object types may be regarded as workflow relevant data which are a part of application data related to and processed by workflow processes (one or more). Apart from standard flow dependencies between activities, also event-based dependencies may be defined. For every atomic activity it is possible to specify that it depends on an *event* and its execution will be postponed until the event is triggered. There are three basic types of events: *external event*, *time event* and *workflow event*. External events are triggered by the workflow environment. Time events and workflow events are triggered by WfM system.

A *compound activity* represents a sub-process and usually is defined to simplify process definition and make use of common activities that exist in many business processes. The last type of activity is a *routing activity* which is used to express control flow elements, namely split and join operations. For both split and join operations three basic control flow operators are defined: *AND, OR,* and *XOR*. On the basic of these operators it is possible to express more complex flow operations. A routing activity is performed by system which is represented by an automatic agent. Since this type of activity is a skeletal activity, which performs no work processing; neither object types nor application is associated with it.

The order of activities within the process is defined by transitions. A *transition* defines the relation which is followed by between two activities. Transition from one activity to another may be conditional (involving expressions which are evaluated to permit or inhibit the transition) or unconditional.

When a workflow process was defined, it may be executed many times. Execution of a workflow process according to its definition is called a *process enactment*. The context of such

enactment includes real performers (workflow participants), concrete relevant data, and specific application call parameters.

The representation of a single enactment of a workflow process is called a *process instance*. Its behaviour is expressed by states and described as a UML statechart diagram. The history of states for a given process instance is represented by *process instance state* class. Every process instance has its own container attributes, may require parameters and provide results. The container attributes are used to control process execution (e.g. in flow conditions). Execution of a process instance may be considered as execution of a set of activity instances. Every activity instance that is an atomic activity is performed by one *workflow participant*. If more than one participant is assigned to the activity, then this activity is instantiated as a set of activity instances with one performer for each of them. Such activity instance may be executed as an *application call* with specific parameters, which operates on *data objects* that are instances of object types assigned to the activity during process definition. If an activity instance is a routing activity it is performed automatically by the system and there is neither application nor data objects assigned to it. In the case when activity instance is a *sub-process*, it is represented by another process instance and executed with appropriate parameters (if required). Similarly to the process instance, behaviour of an activity instance is represented by a UML statechart diagram and stored as activity instance state class.

Flow between activity instances is represented by *transition instances*. When an activity instance is finished the system checks which outgoing transitions may be instantiated. If a transition has no condition or transition condition is satisfied, it is automatically instantiated by the system. A transition instance may be considered as the relation ‘predecessor-successor’ between two activities.

In the next sections all the mentioned workflow entities are described more in detail. Description of every class includes its definition (mostly based on the WfMC’s one), specification of its attributes and information on associations of this class with other metamodel classes.

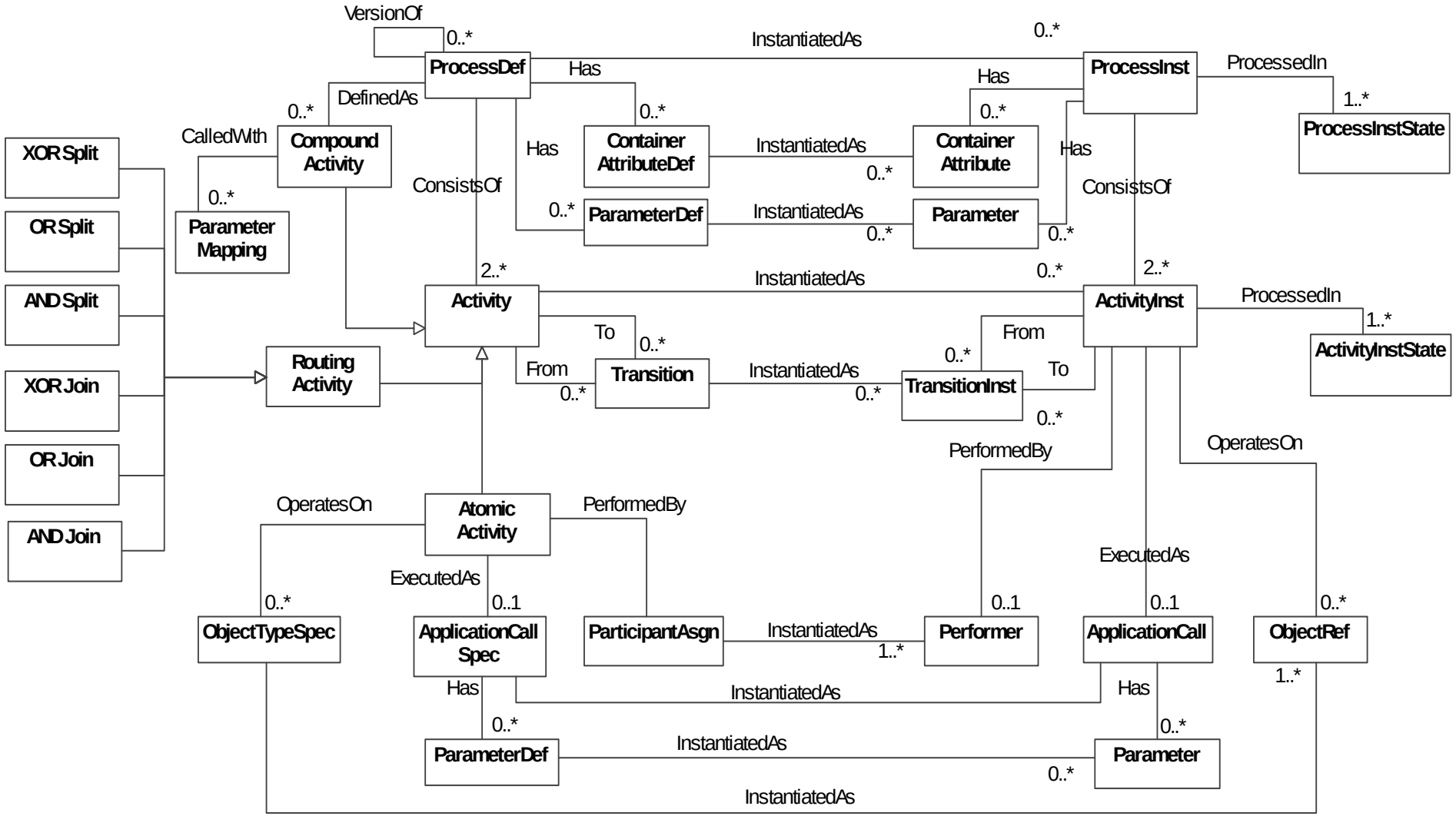

Figure 2.2 The workflow process metamodel – conceptual view

2.4 Required Elements of IT Systems

As stated earlier WfM systems require from IT systems data, services, and resources. For each mentioned element WfM systems need information about its metamodel and its instantiations. Information about the metamodel is used to define workflow processes (i.e. user specification, object type definition, application declaration) while instantiations are used in individual process instances (i.e. particular resources, concrete applications and specific data).

WfM systems keep internally only references to the application services, data and resources. The reference classes defined in the WfM systems together with relations between them and basic application elements are presented in Figure 2.3.

An *object type* is the specification of object invariants that include names and types of attributes, cardinalities, relationships and perhaps other properties. During the use of an application, object types are instantiated as objects. An object possesses a unique identifier and includes values for the attributes specified in the object type.

A *resource definition* describes an application resource. This resource is one of the following types: a user, an automatic agent, and a role. If more complex definition is needed, it may be expressed as a dynamic resource. Such resource is defined as a query which is evaluated during run-time. Resource definition is instantiated as one or more resources (especially in the case of queries).

A *service declaration* specifies the interface of an application. During run-time this interface is used to call the application.

The elements available to WfM systems are just a subset of the data, functional, and resource models defined in IT systems. These models may be much richer in number of classes and associations between them. For instance, a resource model may include the superior-subordinate relation. Such additional information, may be very useful to *take some decisions* during process execution (e.g. select the supervisor of a given user), however *it is not needed to store* such information in the process metamodel.

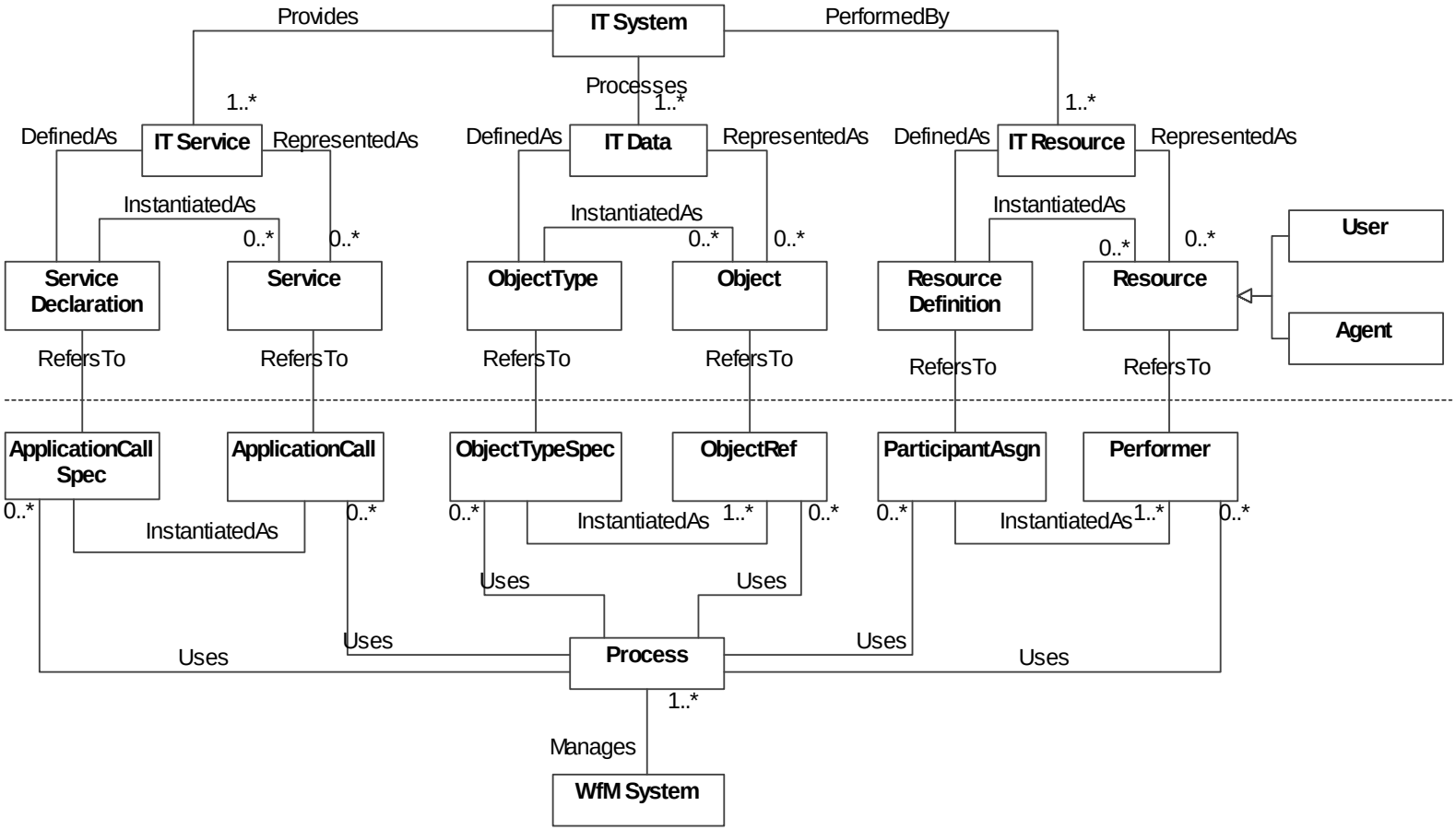

Figure 2.3 Mapping between elements of IT systems and WfM systems – conceptual view

For the workflow monitoring functions which are defined further in the thesis we identified requirements for specification of the attributes only for two entities, namely Resource and User classes. Therefore for these classes we present their attributes and associations more in detail below.

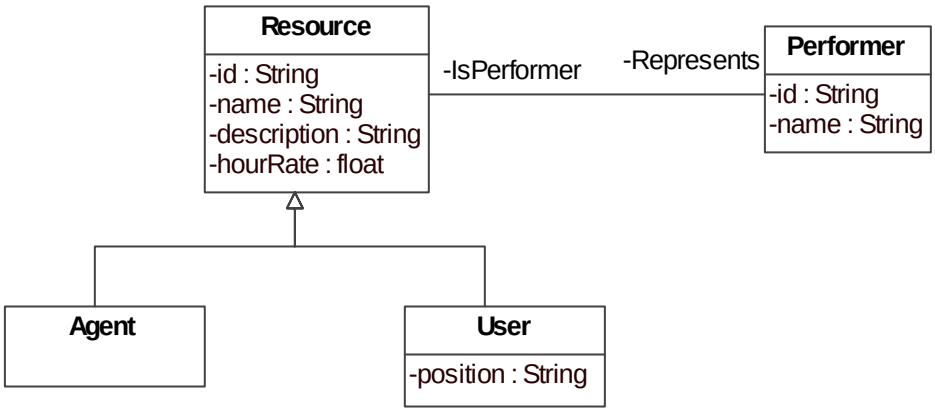

Figure 2.4 Selected elements of the IT system - object database structure view

Table 2.1 Specification of the resource attributes

Attribute Name	Attribute description
id	A unique identifier of the entity at the IT system level.
name	
description	A textual description of the process.

Table 2.2 Associations between resource and performer classes

Association name	Begins: Pointer	Ends: Pointer	Description (ProcessDef view)
RefersTo	Resource: IsPerformer	Performer: Represents	A resource is represented in WfM system as performers.

Table 2.3 Specification of the (additional) user attributes

Attribute Name	Attribute description
position	A position a given user is employed on
hourRate	The cost of one working hour for a given user.

2.5 Process definition

Process definition is a part of the workflow metamodel that is responsible for specification of business processes in a computerised form that may be understood by WfM systems. The next sub-sections describe all the process definition entities. Each description includes entity definition (mostly based on the WfMC's one), specification of its attributes and, if necessary, its behavioural model. A UML class diagram for this part of the metamodel is presented in Figure 2.5.

2.5.1 Process definition

A *process definition* (ProcessDef class) is a representation of a business process in a form, which supports automated manipulation, such as modelling, or enactment by a WfM system. The process definition consists of a network of activities and relationships amongst them, criteria to indicate the start and termination of the process, and information about the individual activities, such as participants, associated applications and data, etc.

Modifications of a business process that occur during its use are considered as new versions of the process definition. It is possible to have many versions at one time, however only one is active (i.e. its current state is *Active*) and may be instantiated. A process definition may represent a part of another process definition and plays the role of its sub-process. It may also have parameters and local attributes (data container).

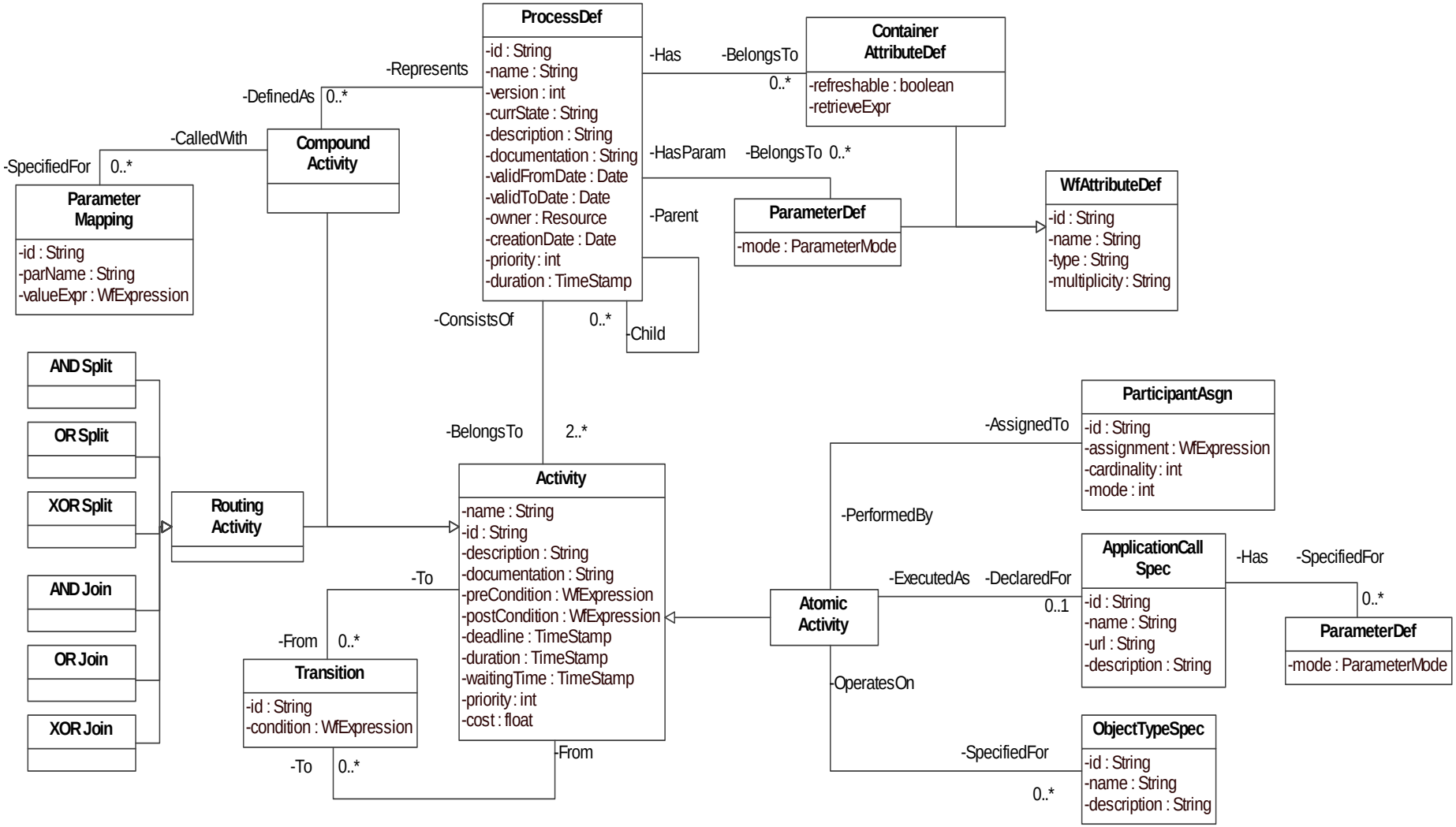

Figure 2.5 The workflow metamodel for the process definition – database schema view

Table 2.4 Specification of the process definition attributes

Attribute Name	Attribute description
id	A unique identifier of the entity at the WfM system level.
name	Different versions of the same process have the same name.
version	A positive integer indicates which version of a given business process is this process definition.
currState	The name (i.e. string value) of the current state according to the behavioural process definition model.
description	A textual description of the process.
documentation	A reference to a file that contains guidelines and suggestions how this process should be performed.
validFromDate	The date from which this process is valid and available to be instantiated.
validToDate	The date to which this process is valid and available to be instantiated.
owner	The user (resource) that is responsible for maintaining this process and all its instances.
creationDate	The date when the process definition has been created.
priority	A positive integer defines the priority of the process. The greater number, the priority is higher.
duration	Expected period required to complete the process. The value is given in standard time units.

Table 2.5 Associations between process definition and other workflow classes

Association name	Begins: Pointer	Ends: Pointer	Description (ProcessDef view)
Manages	WfSystem: Manages	ProcessDef: MangedBy	All process definitions (one or more) are managed by the WfM system.
VersionOf	ProcessDef: Parent	ProcessDef: Child	One process definition (its version) may be used to define one or more new versions of this process.
ConsistsOf	ProcessDef: ConsistsOf	Activity: BelongsTo	A process definition consists of at least two activities.
Has	ProcessDef: Has	ContainerAttributeDef: BelongsTo	A process definition may include one or more attributes.

Has	ProcessDef: HasParam	WfParameterDef: BelongsTo	A process definition may have one or more parameters
InstantiatedAs	ProcessDef: InstantiatedAs	ProcessInst: InstanceOf	A process definition is enacted as process instances.
DefinedAs	CompoundActivity: DefinedAs	ProcessDef: Represents	A process definition may represent a compound activity.

For a process definition the following states within its behavioural model are defined.

Table 2.6 Specification of the process definition states

State	Description
Defined	A process is being defined or modified.
Available.Active	If a defined process has been successfully verified, it becomes active (i.e. changes its state to *Available.Active*). From that moment it may be instantiated for newly created processes. The version of the process that was active before that moment becomes inactive (i.e. changes its state to *Available.InActive*). If we want to modify a process definition which is active, the WfM system creates a new version of the process definition (a copy of the active one) and sets its state to *Defined*.
Available.InActive	A process definition may be used for already instantiated processes but may not be used to create a new instance.

2.5.2 Container attribute definition

Definition of a container attribute (ContainerAttributeDef class) describes data which can be used during process execution to evaluate flow conditions, workflow participant assignment and other elements of the process that depend on such data. In addition to the WfAttributeDef, the container attribute definition includes two attributes: *refreshable* and *retrieveExpr*.

Table 2.7 Specification of the (additional) container attribute definition attributes

Attribute Name	Attribute description
refreshable	It specifies if the value of the attribute has to be re-evaluated before it is used. If the refreshable is set to true, the attribute value is re-evaluated according to the *retrieveExpr*. Otherwise the attribute value is directly used.
retrieveExpr	A WfExpression which specifies how to evaluate a given attribute.

Table 2.8 Associations between container attribute definition and other workflow classes

Association Name	Begins: Pointer	Ends: Pointer	Description (ContainerAttributeDef view)
Has	ProcessDef: Has	ContainerAttributeDef: BelongsTo	A container attribute belongs is defined for a process definition.
InstantiatedAs	ContainerAttributeDef: InstantiatedAs	ContainerAttribute: InstanceOf	A container attribute definition may be instantiated within a process instance.

2.5.3 Activity

An *activity* (Activity class) is a piece of work that forms one logical step within a process. Activity can be atomic, compound or a routing one. The order between activities is defined by transitions. Every activity may have its predecessors and successors.

Table 2.9 Specification of the activity attributes

Attribute Name	Attribute description
id	The successive and unique number of the activity within a given process definition.
name	It is not necessary for the activity name to be unique within a given process.
description	Optional. Description of the activity (textual form).
documentation	Optional. A reference to a file that contains guidelines and suggestions how this activity should be performed, what are its pre and post-conditions, etc.
preCondition	An activity condition expressed as a BPQL query and evaluated when the activity is created. If this condition is satisfied the activity is assigned to the performer. Otherwise, the activity waits until the condition will be satisfied. Such condition can be used, for example, to express that the activity may not be executed before particular date (time constraint).
postCondition	An activity condition expressed as a BPQL query and evaluated when the activity is finished. If it is satisfied the activity is finished. Otherwise, the activity remains in its state and appropriate exception is thrown. The performer can try to finish it later. Such condition can be used, for example, to express that the activity can not be finished by a given date or if a given attribute is empty.
deadline	Time constraint which represents the latest time when, with respecting to

	the process creation date, the activity has to be finished. The value is given in standard time units.
duration	The average time of activity execution. Execution means time between creation of the activity instance (i.e. entering *Running* superstate) and finishing it (i.e. entering *Finished* superstate).
waitingTime	Time constraint which represents the latest time when the activity may wait in the work list queue before it will be executed by the performer. The value of this constraint is given as the percentage of the *duration* value.
priority	Zero or a positive integer. Zero means the lowest priority.
cost	The average cost of activity execution. Zero or a positive float number. Zero means no cost for the activity.

Table 2.10 Associations between activity and other workflow classes

Association name	**Begins: Pointer**	**Ends: Pointer**	**Description (Activity view)**
ConsistsOf	ProcessDef: ConsistsOf	Activity: BelongsTo	An activity belongs to one process definition.
From	Activity: From	Transition: To	An activity may have ingoing transitions.
To	Activity: To	Transition: From	An activity may have outgoing transitions.
InstantiatedAs	Activity: InstantiatedAs	ActivityInst: InstanceOf	An activity may be instantiated within a process instance.

2.5.4 Atomic activity

An *atomic activity* (AtomicActivity class) is typically the smallest (i.e. atomic) unit of work which is scheduled by a workflow manager during process enactment. An atomic activity does not include any additional attribute to those defined for the activity entity.

Table 2.11 Associations between atomic activity and other workflow classes

Association name	**Begins: Pointer**	**Ends: Pointer**	**Description (atomic activity view)**
PerformedBy	AtomicActivity: PerformedBy	ParticipantDef: AssignedTo	An atomic activity is performed by one or more workflow participants specified in

			ParticipantDef.
ExecutedAs	AtomicActivity: ExecutedAs	ApplicationCall Specification: DeclaredFor	An atomic activity is performed *as* an application. Usually, applications are a part of IT systems.
OperatesOn	AtomicActivity: OperatesOn	ObjectType Specification: SpecifiedFor	An atomic activity is performed *on* object types that are also part of IT systems.

2.5.5 Compound Activity

A *compound activity* represents a set of activities which make another process. The reason to separate these activities from the main process definition is that they may be used in more than one workflow process. Such an approach enables an organisation to benefit from re-using of common activities. A sub-process may require some input and output parameters. For every parameter passed to the sub-process it is required to define a mapping from (output one) or to (input one) another workflow objects.

Table 2.12 Associations between compound activity and other workflow classes

Association Name	**Begins: Pointer**	**Ends: Pointer**	**Description (compound activity view)**
DefinedAs	ProcessDef: Represents	CompoundActivity: DefinedAs	A compound activity is represented as a (sub) process.
CalledWith	CompoundActivity: CalledWith	ParameterMapping: SpecifiedFor	A compound activity defines mapping of all sub-process parameters from/to the workflow objects.

2.5.6 Parameter Mapping

A parameter mapping specifies how:

- input process parameters are assigned on the basis of other workflow objects
- output parameters are used to assign the values of other workflow objects (e.g. container attributes).

To define this mapping for both types of parameters we also use BPQL queries.

Table 2.13 Specification of the parameter mapping attributes

Attribute Name	Attribute description
id	A unique identifier of the entity at the WfM system level.
parName	The name of the sub-process parameter.
valueExpr	A BPQL query which represents the mapping.

Table 2.14 Associations between parameter mapping and other workflow classes

Association Name	Begins: Pointer	Ends: Pointer	Description (parameter mapping view)
CalledWith	CompoundActivity: CalledWith	ParameterMapping: SpecifiedFor	A parameter mapping is used for specification of a compound activity based on another process definition.

2.5.7 Routing Activity

A *routing activity* is a control flow element that is used to express alternative, conditional or parallel execution of activities. There are two basic types of route activities, namely: *Split* and *Join* operators.

2.5.7.1 Split Operator

A *split operator* is a routing activity which is used to branch the process flow. It has one ingoing transition and two or more outgoing transitions. In terms of tokens, it consumes one ingoing token and produces one or more outgoing tokens according to the split type. There are three types of split: AND-Split, OR-Split and XOR-Split.

2.5.7.2 Join Operator

A *join operation* is a route activity which is used to merge the process flow. It has two or more ingoing transitions and one outgoing transition. In terms of tokens, it consumes one or more tokens, according to the join type, and produces one outgoing token. There are three types of join: AND-Join, OR-Join and XOR-Join.

2.5.7.3 Parallel routing

Parallel routing enables different activities to be executed in parallel. This type of routing is used to branch and then merge control flow. To model parallel routing two routing operators are identified: AND-Split and AND-Join operators.

AND-Split operator is a routing activity where a single process flow splits into two or more process flows which are executed in parallel. In terms of tokens, AND-Split operator consumes one ingoing token and produces two or more outgoing tokens. The number of tokens is equal to the number of outgoing transitions (i.e. one token for every outgoing transition). The outgoing tokens are processed autonomously and independently until reaching the AND-Join operator.

AND-Join operator is a routing activity where two or more parallel process flows merge into a single process flow. In terms of tokens AND-Join operator consumes all ingoing tokens (exactly one from every ingoing transition) and produces one outgoing token. If the number of outgoing tokens is equal to the count of ingoing transitions (ITR_{count}), the AND-Join operation is completed and the process flow is passed to the next activities. If the number of outgoing tokens is less than ITR_{count}, the AND-Join operation is postponed until the all required tokens (i.e. one from each ingoing transitions) arrive. Finally, if the number of outgoing tokens is greater than ITR_{count}, the AND-Join operator consumes only ITR_{count} tokens, one token from every ingoing transition. Other tokens remain.

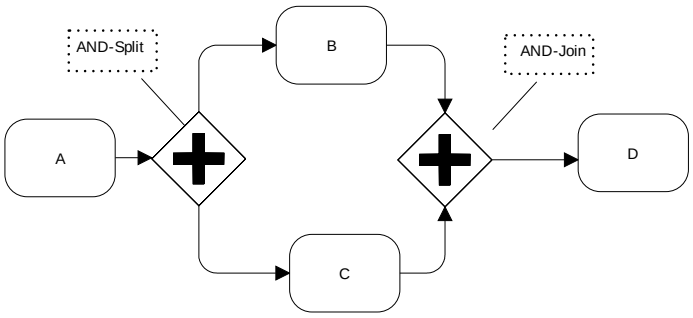

Figure 2.6 Parallel routing example[1]

For instance, in Figure 2.6, when activity A is completed the process flow is passed to the AND-Split operator. This operator consumes the ingoing token and produces two outgoing tokens splitting the process flow into two branches: this with B activity and that with C activity. These activities consume ingoing tokens. Then activities B and C are executed in parallel. This means that B and C are executed at the same time in any order. If one of them is completed, it produces an outgoing token which is passed to AND-Join operator. Since two tokens are required, the AND-Join operator waits until both tokens arrive. When they arrive, they are consumed and the AND-Join operator produces one outgoing token which is passed to activity D.

2.5.7.4 Alternative routing

Alternative routing is used to express an exclusive execution path selection decision which is taken during process execution. In alternative routing one and only one branch from the available branches is satisfied and may be executed (i.e. *either* a *or* b). This is different from parallel routing

[1] In this thesis, we use Business Process Modelling Notation (ver 1.1, May 2004).

where all available branches were executed. To model alternative routing two routing operators are identified: XOR-Split and XOR-Join operators.

XOR-Split operator is a routing activity when a single process flow makes a choice which branch (only one) to take when dealing with multiple alternative workflow branches. For XOR-Split operator every outgoing transition has a condition. Yet, at any point of time condition of one and only one outgoing transition condition is satisfied. In terms of tokens XOR-Split operation consumes one ingoing token and produces one outgoing token. This outgoing token is passed to the activity which is connected with the XOR-Split operation via transition and the condition for this transition is satisfied.

XOR-Join operator is a routing activity when two or more alternative process flows merge into a single process flow. In terms of token, XOR-Join operator consumes one ingoing token and produces one outgoing token. If there are two or more ingoing tokens and there are located in the same ingoing transition then only on of them is consumed and the rest of them remains. If there are two or more ingoing tokens located in two or more ingoing transitions, then this is indicated by WfM system as an exception.

For instance, in Figure 2.7, when activity A is completed the process flow is passed to the XOR-Split operator. This operator consumes one ingoing token and produces one outgoing token. Assuming that x is greater than zero, this token is passed (only) to activity B (i.e. the condition on the transition between the XOR-Split operator and activity B is satisfied). This token is not passed to activity C. The activity B consumes this ingoing token and produces an outgoing token which is then passed to XOR-Join operator. Since this operator waits on one token, it consumes the token and produces a new outgoing token. This token is passed to activity D.

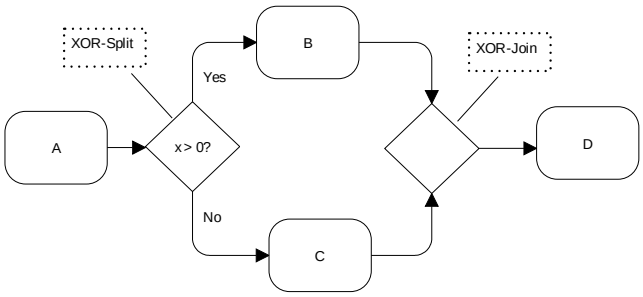

Figure 2.7 Alternative routing example

2.5.7.5 Conditional routing

Conditional routing is used to express a decision which is based on conditional expressions (many possible ways) and is taken during process execution. In conditional routing one or more (also all) branches from the available branches may be satisfied and executed (i.e. a *or* b). To model conditional routing two routing operators are identified: OR-Split and OR-Join operators.

OR-Split operator is a routing activity when a single process flow makes a decision which branches (one or more) to take when encountered with multiple alternative workflow branches. For

the OR-Split operator any outgoing transition has a condition. Yet, at any point of time condition of one or more (also all) outgoing transition conditions may be satisfied. In terms of tokens, OR-Split operation consumes one ingoing token and produces *n* outgoing tokens. The *n* value is equal to the number of outgoing transitions, for which the transition condition is satisfied. These outgoing tokens are passed to the activities which are connected with the OR-Split operation via transitions and the conditions for these transitions are satisfied.

OR-Join operator is a routing activity when two or more conditional process flows merge into a single process flow. In terms of tokens, OR-Join operator consumes all ingoing tokens that were produced upstream (i.e. by activities that are direct or indirect predecessors of a given OR-Join operator) and produces one outgoing token. If any of such tokens have not yet reached the OR-Join operator, this operation is postponed until the all required tokens arrive.

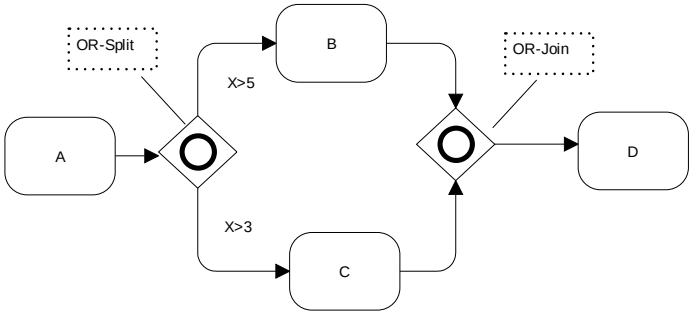

Figure 2.8 The conditional routing example

For instance, in Figure 2.8, when activity A is completed the process flow is passed to the OR-Split operator. This operator consumes one ingoing token and, assuming that x is greater than 5, produces two outgoing token since both transitions (or more precisely transition conditions) are satisfied. These tokens are passed to activities B and C. These activities are executed in parallel. This means that B and C are executed at the same time in any order. If one of them is completed, it produces an outgoing token which is passed to OR-Join operator. Since two tokens are required (both were generated upstream), the OR-Join operator waits until both tokens arrive. When they arrive, they are consumed and the OR-Join operator produces one outgoing token which is passed to activity D.

2.5.8 Transition

A *transition* is a point during execution of a process instance where one activity completes and the thread of control passes to another, which starts. A transition defines the order between two activities.

A *transition* may be unconditional, which means that after completion the 'from' activity the transition is executed and then the activity 'to' starts. If a transition includes a *transition condition*, it means that the transition is executed and then the activity 'to' starts only if the transition condition is satisfied. A transition condition is expressed as a BPQL query.

Table 2.15 Specification of the transition attributes

Attribute Name	Attribute description
id	The successive number of the transition within a given process.
condition	The transition condition expressed as a BPQL query.

Table 2.16 Associations between transition and other workflow classes

Association Name	Begins: Pointer	Ends: Pointer	Description (transition view)
To	Activity: To	Transition: From	A transition starts form an activity.
From	Activity: From	Transition: To	A transition leads to an activity.
InstantiatedAs	Transition: InstantiatedAs	TransitionInst: InstanceOf	A transition may be instantiated within a process instance.

2.5.9 Object Type Specification

Specification of an object type (ObjectTypeSpec class) describes object types which are managed by an IT system and may be used during process enactment. Such specification is expressed by the reference to an object type. Objects of this type may be then used by instantiations of the activity (also applications and performers) to which this specification was assigned.

Table 2.17 Specification of the object type attributes

Attribute Name	Attribute description
id	A unique identifier of the entity at the WfM system level.
name	The name is unique at the IT system level.
description	Textual description of the object type underlying the way how this type is used in the process.

Table 2.18 Associations between object type specification and other workflow classes

Association Name	Begins: Pointer	Ends: Pointer	Description (ObjectTypeSpec view)
OperatesOn	AtomicActivity: OperatesOn	ObjectTypeSpec: SpecifiedFor	Specification of an object type refers to an atomic activity.
InstantiatedAs	ObjectTypeSpec: InstantiatedAs	ObjectRef: InstanceOf	A reference to an instantiation of an object type specification is represented in WfM system as ObjectRef class.

2.5.10 Participant Assignment

Participant assignment (ParticipantAsgn class) specifies workflow participants which are candidates to perform a given activity and have rights to do it. According to the WfMC definition, a workflow participant is one of the following types: a resource, a resource set, a role, an organizational unit, and the system represented by an automatic agent. As an extension to this definition, a role may also be defined as a BPQL query. In this case, the results returned by the query represent workflow participants.

Table 2.19 Specification of the participant assignment attributes

Attribute Name	Attribute description
id	The identifier is unique at the WfM system level.
mode	Determines whether the set of participants will be assigned automatically or manually. It can take one from two values: **Auto** – the system automatically assign workflow participants to a given activity; **Ad-hoc** –the assignment of workflow participants to a given activity is done manually by the performer of the current activity.
cardinality	Determines the number of participants that will perform a given activity. The modifier can take one from two values: **One** – the activity is assigned to the first accepting participant; **All** - the activity is assigned to all the participants returned by a workflow participant assignment.
assignment	A formal definition of the participant assignment. It is expressed as a BPQL query.

Table 2.20 Associations between participant assignment and other workflow classes

Association name	Begins: Pointer	Ends: Pointer	Description (ParticipantAsgn view)
PerformedBy	AtomicActivity: PerformedBy	ParticipantAsgn: AssignedTo	A participant assignment refers to an atomic activity.
InstantiatedAs	ParticipantAsgn: InstantiatedAs	Performer: InstanceOf	The result of the participant assignment evaluation is represented by performer objects.

2.5.11 Application Call Specification

Specification of an application call (ApplicationCallSpec class) describes how a given application will be called, that is what are the input parameters and how to map the output parameters into another workflow objects. The application parameters are defined using *ParameterDef* class.

Table 2.21 Specification of the application call attributes

Attribute Name	Attribute description
id	A unique identifier of the entity at the WfM system level.
name	The name of the application has to be unique at the IT system level.
uri	The (unique) reference to the application.
description	Textual description of the application.

Table 2.22 Associations between application call specification and other workflow classes

Association name	Begins: Pointer	Ends: Pointer	Description (ApplicationCallSpec view)
ExecutedAs	AtomicActivity: ExecutedAs	ApplicationCallSpec: DeclaredFor	Specification of an application call refers to an atomic activity.
InstantiatedAs	ApplicationCallSpec: InstantiatedAs	ApplicationCall: InstanceOf	For activity instance, specification of an application call is instantiated as an application call.
Has	ApplicationCallSpec: Has	WfParameterDef: SpecifiedFor	Specification of an application call may include specification of its parameters.

2.6 Process Enactment

Process enactment is a part of the workflow metamodel that is responsible for execution of business processes according to their process definitions. The next sub-sections describe all the process enactment entities. Each description includes entity definition (mostly based on the WfMC's one), specification of its attributes and, if necessary, its behavioural model. A UML class diagram for this part of the metamodel is presented in Figure 2.9.

2.6.1 Process Instance

A process instance is the representation of a single enactment of a process, or activity within a process, including its associated data. Each process instance represents a separate thread of

execution which may be controlled independently and has its own internal state and externally visible identity. It may be used as a handle, for example, to record or retrieve audit data relating to the individual enactment. A process instance is created, managed and (eventually) terminated by a WfM system, in accordance with the process definition.

Table 2.23 Specification of the process instance attributes

Attribute Name	Attribute description
id	A unique identifier of the entity at the WfM system level.
creationDate	Date when the process instance was created (i.e. entered the *Running* superstate)
finishDate	Date when the process instance was finished (i.e. entered the *Finished* superstate).
currState	The current state according to the process instance behaviour. It is represented as a *ProcessInstState* class.
priority	Zero or a positive integer. Zero means the lowest priority.
Quality of service attributes and constraints	
duration	The current duration of a given process instance. Duration is given in standard time units.
cost	The current cost of execution of a given process instance. Zero or a float number. Zero means no cost for the process.

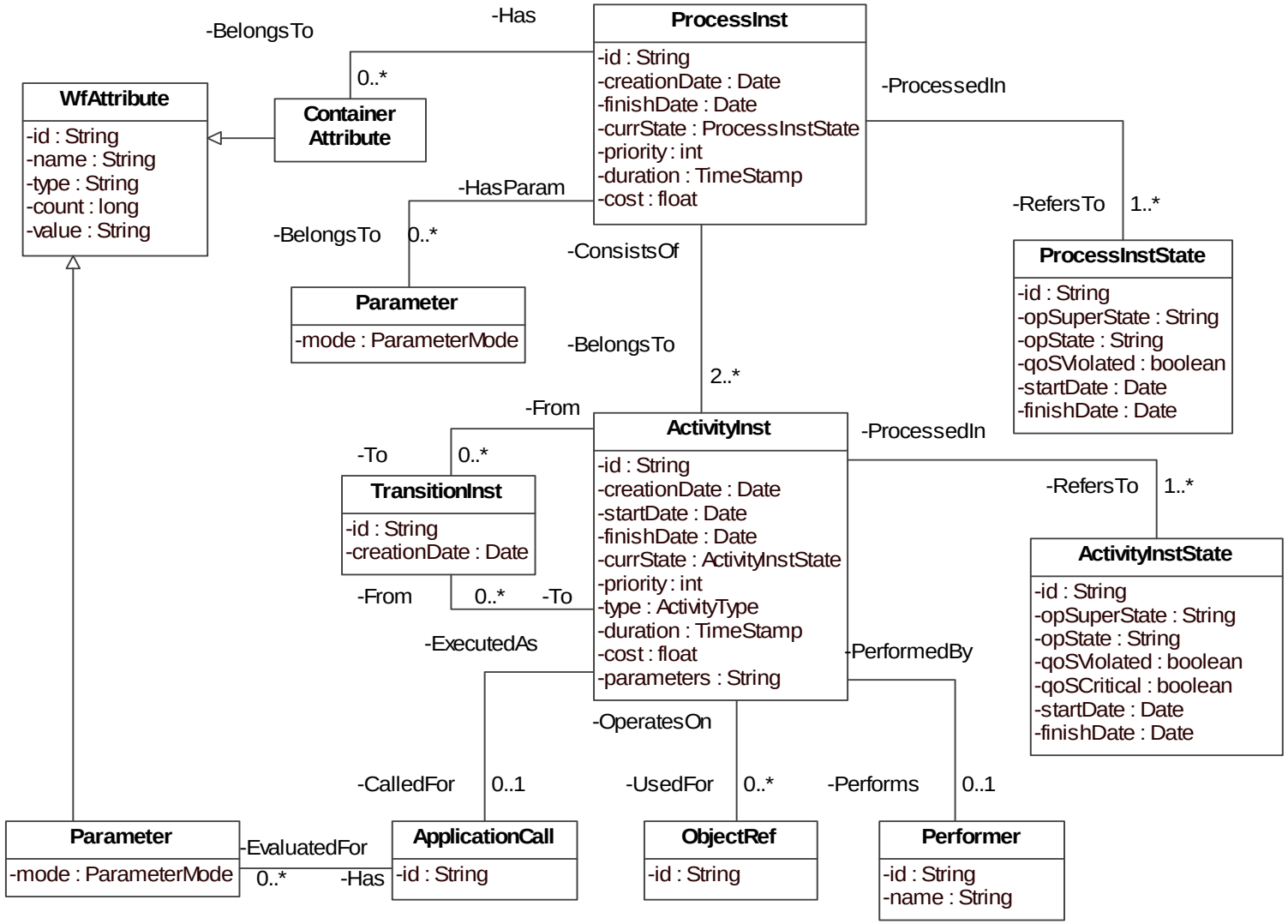

Figure 2.9 The workflow metamodel for the process enactment – database schema view

Table 2.24 Associations between process instance and other workflow classes

Association name	Begins: Pointer	Ends: Pointer	Description (process instance view)
InstantiatedAs	ProcessDef: InstantiatedAs	ProcessInst: InstanceOf	A process instance is created and enacted according to a given process definition.
ProcessedIn:	ProcessInst: ProcessedIn	ProcessInstState: RefersTo	A process instance was/is processed in the states defined within its behavioural model.
ConsistsOf	ProcessInst: ConsistsOf	ActivityInst: BelongsTo	A process instance consists of two or more activity instances.
Has	ProcessInst: Has	ContainerAttribute: BelongsTo	A process instance may have container attributes.
Has	ProcessInst: HasParam	WfParameter: BelongsTo	A process instance may have parameters.

2.6.2 Process Instance State

A process instance state represents information about a state in which a given process instance was or currently remains. The state refers to the process instance behaviour model. A process instance expresses its behaviour in two ways: via operational states and quality of service states. The former states are related to the WfM system operations such as create a process instance, suspend or terminate it.

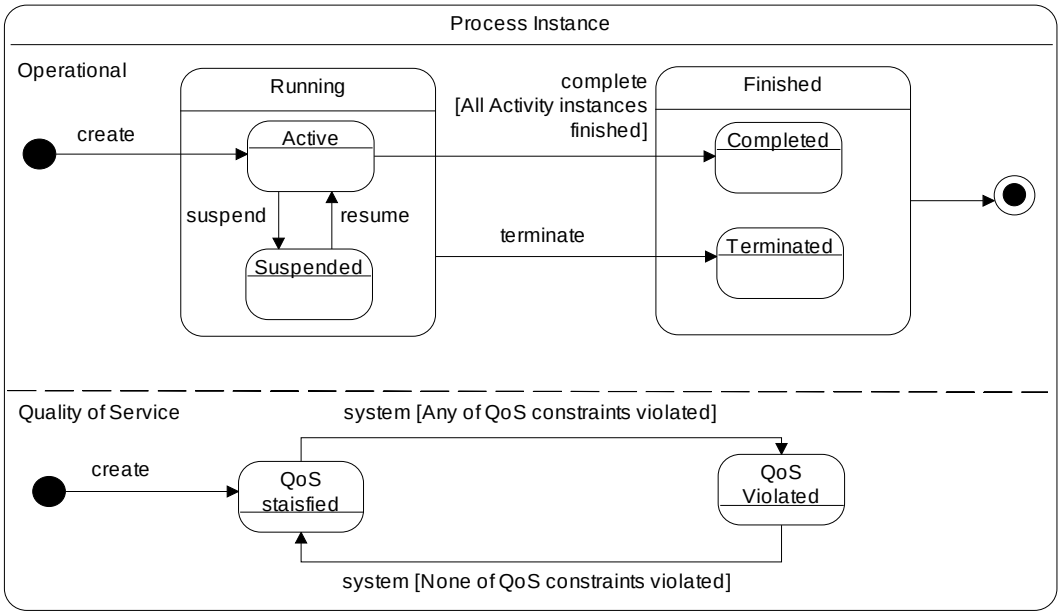

Figure 2.10 Process instance behavioural model

The latter states are related to QoS constraints. They show if all QoS constraints defined for a given process instance are satisfied. The meaning of individual states is described below.

Table 2.25 Process instance state description

State	Description
Operational states	
Running	A superstate which represents situation when a given process instance has been created.
Running.Active	Execution of the process instance has already started.
Running.Suspended	The process instance is quiescent; no further activities are started until it is resumed.
Finished	A super-state which represents situation when a given process instance has been finished (completed or terminated).
Finished.Completed	All activity instances belonged to the process instance have been completed (i.e. entered the *Finished* superstate).
Finished.Terminated	Execution of the process has been stopped (abnormally) due to error or user request.
QoS states	
QoS satisfied	All of the QoS constraints defined for a given process instance are currently satisfied.
QoS violated	At least one of the QoS constraints defined for a given process instance is violated.

Table 2.26 Specification of the process instance state attributes

Attribute Name	Attribute description
id	A unique identifier of the entity at the WfM system level.
opSuperState	The name of the operational superstate specified in the process instance behavioural model.
opState	The name of the state within a given operational superstate specified in the process instance behavioural model.
qoSViolated	True if all the QoS constraints for a given process instance are satisfied, otherwise – false.
startDate	The date when the process instance entered the state.
finishDate	The date when the process instance left the state. For the current state this attribute is zero.

Table 2.27 Associations between process instance state and other workflow classes

Association name	Begins: Pointer	Ends: Pointer	Description (process instance state view)
ProcessedIn	ProcessInst: ProcessedIn	ProcessInstState: RefersTo	A process instance state refers to a process instance describing its dynamics.

2.6.3 Container Attribute

A *container attribute* is a piece of data used during execution of a given process instance to evaluate flow conditions, workflow participant assignment and other elements of the process that depend on such data. It has the same attributes as the *WfAttribute* class.

Table 2.28 Associations between container attribute and other workflow classes

Association name	Begins: Pointer	Ends: Pointer	Description (container attribute view)
Has	ProcessInst: Has	ContainerAttribute: BelongsTo	A container attribute belongs to a process instance.

2.6.4 Activity Instance

An *activity instance* represents an activity within a (single) enactment of a process, i.e. within a process instance. An activity instance is created and managed by a WfM system when required within the enactment of process, in accordance with the process definition.

Each activity instance represents a single invocation of an activity, relates to exactly one process instance and uses the process instance data associated with the process instance. Several activity instances may be associated with one process instance, where parallel activities exist within the process, but one activity instance cannot be associated with more than one process instance.

Table 2.29 Specification of the activity instance attributes

Attribute Name	Attribute description
id	A unique identifier of the entity at the WfM system level.
creationDate	The date when the process instance enters the *Created* superstate.
startDate	The date when the process instance enters the *Running* superstate.
finishDate	The date when the process instance enters the *Finished* state.
currState	The current state according to the activity instance behaviour. It is represented as a *ActivityInstState* class.
priority	Zero or a positive integer. 0 means the lowest priority.
type	An *ActivityType* value specifying if the activity instance is an atomic

	(type= ATOMIC), routing (type=ROUTING) or a compound one (type = COMPOUND).
duration	The current duration of a given activity instance. Duration is given in standard time units.
waitingTime	The current period which a given activity instance has spent in the *Created.Scheduled* state. Waiting time is given in standard time units.
cost	The current cost of execution of a given activity instance. Zero or a float number. Zero means no cost.
deadline	Time constraint defines the latest time when, with respecting to the process creation date, the activity instance has to be finished (i.e. changes its state to *Finished.**). A deadline is given in standard time units.

Table 2.30 Associations between activity instance and other workflow classes

Association Name	**Begins: Pointer**	**Ends: Pointer**	**Description (activity instance view)**
ConsistsOf	ProcessInst: ConsistsOf	ActivityInst: BelongsTo	An activity instance belongs to a process instance.
InstantiatedAs	Activity: InstantiatedAs	ActivityInst: InstanceOf	An activity instance is created and enacted according to a given activity (definition). All three activity types, namely atomic, routing and compound ones are instantiated as activity instance.
ProcessedIn	ActivityInst: ProcessedIn	ActivityInstState: RefersTo	An activity instance was/is processed in the states defined within its behavioural model.
From	ActivityInst: From	TransitionInst: To	An activity instance may have outgoing transition instances.
To	ActivityInst: To	TransitionInst: From	An activity instance may have ingoing transition instances.
PerformedBy	ActivityInst: PerformedBy	Performer: Performs	This association is valid for atomic and routing activities. For compound activities there is no performer associated. Every atomic activity is

			performed by one performer. If more than one workflow participants were selected in workflow participant assignment then the appropriate number of activity instances is created – for every performer. Routing activities are performed by the WfM system represented by an automatic agent.
ExecutedAs	ActivityInst: ExecutedAs	ApplicationCall: CalledFor	This association is valid for atomic and compound activities. For routing activities there is no application associated. For compound activities the application call is represented as a call to another process (sub-process). Such call is a request for creation a new instantiation of this sub-process. In addition, the values of the input and output parameters of the sub-process are set or extracted according to the compound process mapping rules (see ParameterMapping class).
OperatesOn	ActivityInst: OperatesOn	ObjectReference: UsedFor	This association is valid for atomic activities only. It shows what objects available as relevant data are created, updated, deleted or used within execution of a given activity.

2.6.5 Activity Instance State

An *activity instance state* (ActivityInstState class) represents information about a state in which a given activity instance was or currently remain. The state refers to the activity instance behaviour

model. An activity instance expresses its behaviour in two ways: via operational states and Quality of Service states. The former states are related to the WfM system operations such as create an activity instance, suspend or terminate it.

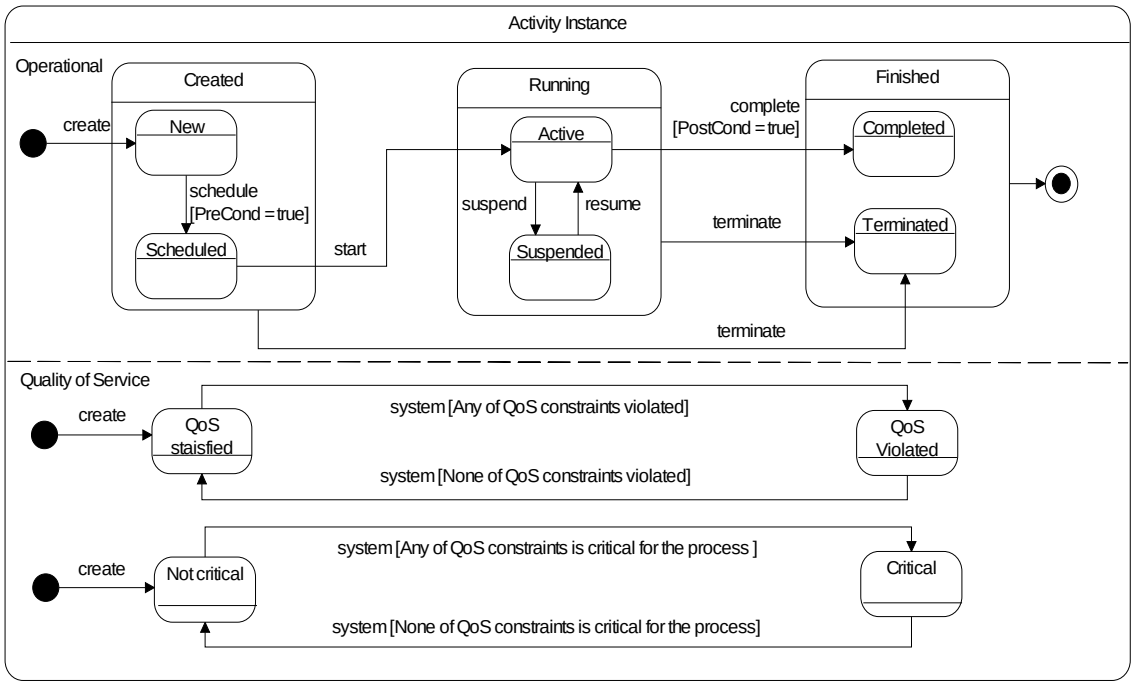

Figure 2.11 Activity instance behavioural model

The latter states are related to QoS constraints. They show if all QoS constraints defined for a given activity instance are satisfied as well as checks if any of these QoS constraints are critical for the whole process. The meaning of individual states is described below.

Table 2.31 Activity instance state description

State	Description
Operational states	
Created	A superstate which represents situation when a given activity instance was created but has not been executed yet by the performer.
Created.New	The activity instance has been created. Its pre-condition has not been verified yet, or is not satisfied. If there is no pre-condition defined, the activity is placed automatically in the state *Created.Scheduled*.
Created.Scheduled	The pre-condition for the activity instance has been satisfied, assigned to a performer and started successfully. However, so far, the activity instance has not been executed by the performer.
Running	A superstate which represents situation when a given activity was started and is executed by the performer.
Running.Active	The activity instance represented by a workitem has been taken by the

	performer and is currently executing.
Running.Suspended	The activity instance represented by a workitem has been taken by the performer but is not currently executing.
Finished	A superstate which represents situation when a given activity instance has already been finished (completed or terminated).
Finished.Completed	The activity instance has been finished correctly.
Finished.Terminated	The activity instance has been stopped (abnormally) due to error or user request.
QoS states	
QoS satisfied	All of the QoS constraints defined for a given activity instance are currently satisfied.
QoS violated	At least one of the QoS constraints defined for a given activity instance is violated.
Not Critical	None of the QoS constraints defined for a given activity instance is critical for the whole process.
Critical	At least one of the QoS constraints defined for a given activity instance is critical for the whole process.

Table 2.32 Specification of the activity instance state attributes

Attribute Name	Attribute description
id	A unique identifier of the entity at the WfM system level.
opSuperState	The name of the operational superstate specified in the activity instance behavioural model.
opState	The name of the state within a given operational superstate specified in the activity instance behavioural model.
qoSViolated	True if any of the QoS constraints is violated, otherwise – false.
qoSCritical	True if any of the QoS constraints is critical for the whole process, otherwise – false.
startDate	The date when the process instance entered the state created.
finishDate	The date when the process instance left the state. For the current state, this attribute is zero.

Table 2.33 Associations between activity instance state and other workflow classes

Association Name	Begins: Pointer	Ends: Pointer	Description (process instance state view)
ProcessedIn	ActivityInst: ProcessedIn	ActivityInstState: RefersTo	An activity instance state refers to an activity instance describing its dynamics.

2.6.6 Transition Instance

A *transition instance* represents a single connection between two activity instances.

Table 2.34 Specification of the transition instance attributes

Attribute Name	Attribute description
id	A unique identifier of the entity at the WfM system level.
creationDate	The date when transition was fired.

Table 2.35 Associations between transition instance and other workflow classes

Association Name	Begins: Pointer	Ends: Pointer	Description (transition view)
InstantiatedAs	Transition: From	TransitionInst: To	A transition instance is created and enacted according to a given transition (definition).
From	ActivityInst: From	TransitionInst: To	A transition instance starts from an activity instance.
To	TransitionInst: From	ActivityInst: To	A transition instance ends at an activity instance.

2.6.7 Object Reference

An *object reference* (ObjectRef class) is an instantiation of a data type (object) related to a given activity. This object is a part of the application relevant data and used by the activity instance performer or application to execute the activity instance.

Table 2.36 Specification of the object reference attributes

Attribute Name	Attribute description
id	A unique identifier of the entity at the WfM system level.

Table 2.37 Associations between object reference and other workflow classes

Association Name	Begins: Pointer	Ends: Pointer	Description (ObjectRef view)
InstantiatedAs	ObjectTypeSpec: InstantiatedAs	ObjectRef: InstanceOf	An ObjectRef refers to an instantiation of an object type specified for a given activity within the process definition.
OperatesOn	ActivityInst: OperatesOn	ObjectRef: UsedFor	For atomic activities an ObjectRef refers to an object which is used by the activity instance performer to execute it.

2.6.8 Performer

A *performer* is a workflow participant that has performed, is performing or will perform a given activity instance. For an activity instance only one performer is assigned. If two or more workflow participants were assigned to a given activity, appropriate number of activity instances is created.

Table 2.38 Specification of the object reference attributes

Attribute Name	Attribute description
id	A unique identifier of the entity at the WfM system level.
name	If the associated activity instance is a routing activity or an automatic atomic activity, the name is ‘system’.

Table 2.39 Associations between object reference and other workflow classes

Association Name	Begins: Pointer	Ends: Pointer	Description (Performer view)
InstantiatedAs	ParticipantAsgn: InstantiatedAs	Performer: InstanceOf	A performer is a workflow participant selected from a list of candidates to perform a given activity. The candidates are determined according to the participant assignment defined within the process definition.
PerformedBy	ActivityInst: PerformedBy	Performer: Performs	A performer performs an activity instance.

2.6.9 Application Call

An *application call* (ApplicationCall class) gives a reference to the application (atomic activity) or sub-process (compound activity) that has been executed on behalf of a given activity. The parameters of the call are represented as a collection of WfParameter objects.

Table 2.40 Specification of the application call attributes

Attribute Name	Attribute description
id	A unique identifier of the entity at the WfM system level.

Table 2.41 Associations between application call and other workflow classes

Association Name	Begins: Pointer	Ends: Pointer	Description (Performer view)
InstantiatedAs	ApplicationCallSpec: InstantiatedAs	ApplicationCall: InstanceOf	An application call is an instantiation of the application call specification.
ExecutedAs	ActivityInst: ExecutedAs	ApplicationCall: CalledFor	An application call with particular parameters is executed on behalf of a given activity instance
Has	ApplicationCall: Has	WfParameter: EvaluatedFor	An application cal may have parameters.

2.7 Basic Workflow Types

This section describes the workflow entities that are used or inherited by the other workflow entities included in the metamodel.

2.7.1 Attribute Definition

An attribute definition (WfAttributeDef class) is used to specify an attribute or a parameter for some other workflow classes such as process definition or specification of an application call.

Table 2.42 Specification of the Workflow Attribute Definition attributes

Attribute Name	Attribute description
id	The unique identifier at the level of the WfM system.
name	The attribute name has to be unique at the process level.
type	The attribute may be one of the following types: Boolean, integer, float, string, date and an XML document. The last type may be used to

	represent a compound attribute. Access to the individual elements of such compound attribute may be done by using an XPath expression. In addition, WfAttribute type may include a set of values of one of the mentioned types. In this case the multiplicity represents the restrictions on the number of such values.
multiplicity	The possible number of data values for the attribute that may be held by an instance. The cardinality of the set of values is an implicit part of the attribute. In the common case in which the multiplicity is 1..1, then the attribute is a scalar (i.e., it holds exactly one value). • 0 - zero and only zero. • 1 - one and only one. • 0..1 - zero or one. • 0..* - from zero to any positive integer. • 1..* - from one to any positive integer. • * - any positive integer.

2.7.2 Attribute

The attribute class (WfAttribute class) is used in many workflow objects to represent the actual values of their attributes.

Table 2.43 Specification of the Workflow Attribute attributes

Attribute Name	Attribute description
id	The unique identifier at the level of the WfM system.
name	The attribute name has to be unique at the process level.
type	The attribute type according to the attribute specification.
count	Specifies the actual number of the single values assigned to the attribute. Important for the attributes with multiplicity different than 1.
value	The value of an attribute is kept as a string and if used converted into the right type.

2.7.3 Parameter definition

A parameter definition (WfParameterDef class) is used to specifies parameters for a process definition and an application call. There are two types of parameters available: input parameters and output parameters. The input parameters are the arguments which are passed when the called entity (i.e. process or application) starts. The output parameters are the called entity results assigned when it completes.

The *WfParameteDef* class is a specialisation of the *WfAttributeDef* class. In addition to the latter class, the parameter definition includes two attributes: *mode* and *evalSpec*.

Table 2.44 Specification of the (additional) parameter definition attributes

Attribute Name	Attribute description
mode	A WfParameterMode value specifying if the parameter is an argument (mode = IN) or a result (mode = OUT).
evalSpec	A BPQL query which represent a way how: • an input parameter is mapped into other workflow objects within a given process (e.g. container attributes or application parameters) • an output parameter is evaluated on the basis of the other workflow objects.

Table 2.45 Associations between parameter definition and other workflow objects

Association Name	Begins: Pointer	Ends: Pointer	Description (WfParameterDef view)
Has	ProcessDef: HasParam	WfParameterDef: BelongsTo	A parameter is defined for a process definition.
Has	ApplicationCallSpec: Has	WfParameterDef: BelongsTo	A parameter is defined for an application.
InstantiatedAs	WfParameterDef: InstantiatedAs	WfParameter: InstanceOf	A parameter definition may be instantiated within a process instance as its parameter or as a parameter of an application call within a given activity.

2.7.4 Parameter

A parameter (WfParameter class) is used to represent a parameter of process instance or an application call. In addition to the WfAttribute, the parameter definition includes one attribute, namely *mode*.

Table 2.46 Specification of the (additional) parameter attributes

Attribute Name	Attribute description
mode	A WfParameterMode value specifying if the parameter is an argument (mode = IN) or a result (mode = OUT).

Table 2.47 Associations between parameter and other workflow objects

Association Name	Begins: Pointer	Ends: Pointer	Description (WfParameterDef view)
Has	ProcessInst: HasParam	WfParameter: BelongsTo	A process instance may have parameters.
Has	ApplicationCall: Has	WfParameter: BelongsTo	An application call may have parameters.
InstantiatedAs	WfParameterDef: InstantiatedAs	WfParameter: InstanceOf	A parameter is an instantiation of a parameter definition.

2.7.5 Workflow Expression

A workflow expression (WfExpression class) is a textual specification how workflow objects are assigned or extracted. Workflow expressions operate on workflow control data and may use workflow relevant data, operators and functions. They are defined during the design of workflow processes and evaluated during process enactment and monitoring when given workflow objects have to be assigned or retrieved. Their definition is written in a query language that is able to operate on the workflow metamodel.

Workflow expressions may be considered as a flexible and universal way to express complex and dynamic dependencies on workflow objects that needs to be evaluated in runtime. In this thesis we propose that the workflow expressions should be defined using Business Process Query Language which is described more in detail in the next chapters.

Table 2.48 Specification of the workflow expression attributes

Attribute Name	Attribute description
id	The unique identifier at the level of the WfM system.
expr	A textual representation of the workflow expression.
langType	The type of the expression language used to define the expression. By default this is 'BPQL'.

2.7.6 Workflow Expression Evaluation

A workflow expression evaluation (WfExpressionEval class) is to represent the context of the expression evaluation as well as to collect the evaluation results. The context is expressed by a restricted set of the following parameters which may be used to evaluate the query:

- **process instance identifier** which the query operates on. It is then used in the *ThisProcessInst* function to extract the appropriate process instance.
- **activity instance identifier** which the query operates on. It is then used in the *ThisActivityInst* BPQL function to extract the appropriate activity instance.

The above parameters are optional. If they are not set but a query requires to extract their values the WfM system signalises an error.

Table 2.49 Specification of the workflow expression evaluation attributes

Attribute Name	Attribute description
id	The unique identifier at the level of the WfM system.
procesInstId	Identifier of the process instance which the query operates on.
activityInstId	Identifier of the activity instance which the query operates on.
result	A collection of values returned by the query.

2.7.7 Parameter Mode

This entity (WfParameterMode enumeration) is a dictionary which specifies the available parameter modes. There are two parameter modes defined: IN which means input parameter and OUT which means output parameter.

2.7.8 Activity Instance Type

This entity (ActivityInstType enumeration) is a dictionary which specifies the available activity instance types. There are three types defined: ATOMIC, ROUTING and COMPOUND.

2.8 Closing Remarks

We have described to some level of detail basic classes and attributes of the workflow metamodel. Our specification directly stems from the text of the official WfM standards. Because the standards do not specify the metamodel precisely, there are a lot of free choices. This part of the thesis has consciously been written in the style of these standards, as we hope that after some iterations it can be accepted by corresponding bodies (e.g. WfMC) as a reference metamodel. For this reason the specification has the flavour of industrial project documentation rather than high-level scientific theoretical assumptions. Although the presented material can be perceived as too simplistic and formalistic, we believe this style is absolutely necessary to convince business-oriented parties that we propose a valuable solution.

There are many potential workflow metamodel features that are not taken into account in the above description. Further metamodel properties that can be useful in many workflow monitoring cases concern states of documents processed by workflow processes, states of vital resources (money, time, manpower, equipment, office infrastructure, etc.), statistics and datamining results, some reports (e.g. total work done by workflow participants for last month), and so on. They will be the subject of further research and refinement.

3 Business Process Query Language Foundations

In the previous chapter we have defined the workflow process metamodel, which specifies basic workflow objects and associations between them on a conceptual level. In order to extract information from the data organised according to this metamodel, it is necessary to define a query language. The language is further referred to as Business Process Query Language (BPQL).

Usually query languages are not oriented towards a particular application domain. Hence the need for a special query language for a workflow metamodel is not obvious from the point of view of the language definition. However, the need for such a language becomes more evident when we come to the standardisation and implementation level. Usually it is not enough to say the workflow programmers can use e.g. SQL or OQL, because these languages are not associated with the workflow environment in the sense of workflow standards. The need for a special query language for workflow environments stems from at least three reasons:

- The workflow community, including standardisation bodies, has to be convinced that the proposed query language satisfies their needs;
- There can be special implementation issues, such as some specialized query operators or the methods of embedding a query language into some process definition language, e.g. XPDL;
- The language should be equipped in a library of special procedures, functions, methods and views that will present an essential support of the workflow programmers and will be dedicated to the assumed workflow metamodel.

The choice of a query language is constrained by many criteria. The most important of them concerns the choice of a datamodel that will be used for implementation of the workflow metamodel. In the previous chapter we present the metamodel in terms of an object model similar to UML. From the conceptual point of view this is the most reasonable choice. In such a case our next decisions are constrained to object-oriented query languages, such as ODMG OQL [ODMG00] or SBQL [Subieta04]. If we assume object-relational datamodel then we are constrained to SQL-99 or SQL-2003. Assuming XML/RDF datamodels we are constrained to XML or RDF query languages such as XQuery [XQuery03]. In this context SBQL is a candidate as well, since it is defined also for XML/RDF datamodels. Assuming that the target metamodel implementation will be based on a relational database we are constrained to some variant of SQL,

e.g. SQL-89, SQL-92 or SQL implemented in particular relational DBMS such as Oracle PL/SQL, MS SQL Server, IBM DB2, Sybase or Informix.

Moreover, the choice is constrained by other factors such as the precision of specification of semantics, popularity of a particular language, or the effort necessary for implementation. In particular, it is a common belief that ODMG OQL specification of semantics is very imprecise and sometimes inconsistent thus coherent implementation can be problematic. Similarly, many professionals claim that specification of SQL-99 is so huge that no company can entirely implement it. In fact there are many contradictory criteria concerning the choice of a query language for a workflow metamodel and some systematic taxonomy does not exist. Additional difficulty with query languages' taxonomies concerns if we are talking about some theoretical proposals (e.g. standards such as SQL-92, SQL-99 or ODMG OQL) or languages implemented in particular systems. As a rule, theoretical proposals are not implemented entirely or are implemented with a lot of essential differences. Many practical proposals are in turn proprietary for particular systems, e.g. SQL in Oracle is much different than SQL in IBM DB2. In general, the choice of a data model and a corresponding query language is under influence of some market trends, beliefs or predictions of particular communities rather than by some measurable criteria.

In the following we present requirements for BPQL assuming that the choice is not constrained by other vital factors such as an implementation tool used for the workflow metamodel. The list of these requirements includes workflow specific requirements as well as general query language requirements. This list is then used to define BPQL foundations concerning business process/workflow management and query languages. We discuss whether BPQL should initiate a new workflow management framework (revolution) or rather enrich existing ones of new features and functionality making them more flexible (evolution). We examine whether BPQL should be developed from scratch or to be a clone of a well-known query language. We also include a comparative study of existing query languages considered as candidate patterns for BPQL. Due to huge number of existing query languages our comparison will take into account only some of them which we consider the most promising ones.

3.1 Requirements for BPQL

This section includes a list of the main requirements for BPQL. These requirements have been collected by over four years on the basis of (potential) customers needs as well as modern workflow management trends.

Req. 1 Clear, simple and expressive syntax

Syntactic issues are usually neglected in the scientific literature, but in practice the language syntax determines to a big extent usability of the language. Usually too complex syntax makes the

language hard to understand and apply. Yet, many language constructs give opportunity to introduce language redundancy and anomalies. (to compare see [Melton01], [Melton93]). The syntax should be *expressive* and able to identify:

- any workflow object included in the metamodel,
- any attribute for any mentioned workflow objects,
- any association between two or more mentioned workflow objects. All types of associations should be supported: one-to-one (e.g. assignment of an application to a given activity), one-to-many (e.g. instantiation of a process definition) and many-to-many (order dependencies between activities),
- any value that belongs to one of the metamodel dictionaries (e.g. the name of activity instance states, type of routing activity).

Moreover, the BPQL syntax should express every query that makes sense for the users. This property of the language is frequently referred to as *orthogonality* or *compositionality*. 'Making sense' concerns both humans and automatic program generators.

Req. 2 Unambiguous, coherent and complete semantics

BPQL semantics should be *unambiguous*. First of all, this feature is crucial for application of BPQL. If any language construct is ambiguous, it means that we do not know in which way it will be interpreted/processed. This feature is also crucial for optimisation of BPQL queries when one query is transformed to another query for which the execution cost seems to be smaller. Lack or incomplete set of optimisation features is the first symptom that the language may be inefficient.

BPQL semantics needs to be *coherent*. Lack of coherency may introduce faults which make consistent implementation of the language processor impossible. There are many incoherent features of current query languages (e.g. ODMG OQL; see [Alagic97, Subieta97]).

BPQL semantics has also to be *complete*. From the application point of view, if a language provides some construction without proper semantics defined, usually such constructions are hardly used and/or implemented in different way by various development teams. In practice such situation may cause that the language will be too difficult to implement and thus not applicable. For instance, if the meaning of the "auxiliary variable" is not defined precisely, the concept will be understood differently in different context, resulting in many different implementations even in the same language. Again, the example is from ODMG OQL, where some "variable" concept is imprecisely defined in 8 different contexts: in "from" clauses, as a "free variable" of quantifiers, in "order by" clauses, in "group by" clauses, etc.

Req. 3 Pragmatic universality, including a rich set of basic operators and built-in functions

The debate on universality of query languages is continued from 1970-ties. At that time many people believed in the concept of "relational completeness" as a proper measure of the query

languages' universality. The relational completeness was understood as the power of the relational algebra or calculus. However, SQL includes a lot of features not expressible in the relational algebra, such as arithmetic operators, aggregate functions, grouping, ordering, etc. Still, there are a lot of queries that are not expressible in SQL, for instance, queries addressing Bill-Of-Material (BOM) hierarchies, which require advanced recursive capabilities [Atkinson87]. The comparison of relational completeness with the power of SQL has led many professional to the conclusion that the "relational completeness" is barely a random, poorly motivated point on the full scale of universality of query languages. Some scientists have started to believe that the proper yardstick of query languages universality is the power of the Turing machine (Turing completeness). This argument is frequently used by the community dealing with logic-oriented query languages such as Datalog [Ceri89]. Unfortunately, this measure is compromised by the fact that Turing completeness is very easy to achieve; in fact it is enough to retain some 10% of the Basic programming language functionality to have the power of the Turing machine. The measure is totally irrelevant to SQL and similar query languages. Probably, it is rather easy to achieve Turing completeness in SQL, but this makes little sense and does not change anything in the usability of the language.

Moreover, universality of a query language should also take into account the datamodel that is addressed by a query language. The universality would mean that for any mapping of data structures defined according a particular datamodel there is a corresponding query. In such terms no language is universal. In particular, SQL-89 and SQL-92 address the relational model only and are irrelevant to more complex databases, which include e.g. complex hierarchical objects. Similarly, XQuery addresses only hierarchical XML-like data structures and does not deal with classes, methods, inheritance, polymorphism, associations among objects, encapsulation, etc. SQL-99 and SQL-2003 address an own object-relational datamodel with unknown power and limitations. Various dialects of Datalog address structures similar to relational ones, sometimes extended by particular features, e.g. complex objects. So far there is no Datalog dialect addressing classes, methods, inheritance, polymorphism, dynamic object roles, encapsulation and other features. ODMG OQL addresses an own "pure" object model, but there are also limitations, e.g. the model does not include database views, dynamic object roles, dynamic inheritance, encapsulation, etc. SBQL addresses the most universal family of data models, which covers relational, object-relational, XML-oriented and object-oriented datamodels, including classes, methods, views, static inheritance, dynamic inheritance, encapsulation, etc. but still has limitations, e.g. it does not deal with n-ary associations. Various incompatible features that are currently proposed for datamodels cause the definition of universality of query languages extremely difficult or (most probably) impossible.

Thus currently there are no more attempts to define the universality of query languages in mathematical terms. Probably such a measure does not exist or its definition would have no meaning for practice. The reasonable definitions must refer to the *use* of the language by its users.

Pragmatic universality of a query language means that the programmer is able to accomplish all (or *almost* all) the tasks from the application domain using reasonable effort for producing efficient, reliable and easy to maintain software. As follows from the definition, the measure is subjective and relative to the state-of-the-art in the domain of query/programming languages, to definitions of "reasonable effort", and other qualities of the query/programming environment.

Hence BPQL should be comparable concerning query operators with other query languages such as SQL, OQL, XQuery and SBQL. It should also be equipped with a rich set of operators proprietary to business process definition languages. This set should include basic algebraic as well as non-algebraic operators (in the Stack-Based Approach terms [Subieta04]). There are a lot of algebraic operators necessary in query languages, such as numerical and string operators and comparisons, Boolean operators, aggregate functions, union, intersection, a conditional expression (if-then-else), aliasing (*as* operator), grouping, and others. As non-algebraic basic operators the following ones should be supported:

- *Selection* (where) which makes it possible to extract only those objects that satisfy a given condition. For instance, to get a list of activity instances that belong to a given process instance, we should use a selection operator followed by a condition on the process instance identifier. Following SBA, the operator should be generalized in comparison to the above description.
- *Projection* and *navigation* (including path expressions) which may be used either to extract information about selected attributes of workflow object or to operate on various objects related to each other by associations. For instance, when we need information about the date when a given activity instance was created, we extract only the attribute *createDate* from all the attributes it possesses. In addition, if we want to extract information about the previous activities of the current activity, we need first to extract information about the association between the activity and its predecessors, and then, from such association, extract information about the activity predecessors. Following SBA, the operator should be much generalized.
- *Dependent join* which associates objects with other objects. For instance, if we want to extract information about the activity name (i.e. activity object) and number of its instances (selection and count on activity instance objects), we get a new object with two attributes: activity name and count of its present instances. In comparison to ODMG OQL and SQL-99 in SBA the operator should be generalized and should be orthogonal to other operators.
- Quantifiers – both *universal* and *existential quantifiers* should be provided. These operators are very useful when population of objects is considered. It is especially true, when it is required to check whether one, some, or all selected objects satisfy a given condition. Following SBA, the operators should be generalized in comparison to the current query languages.

- Sorting (order by) operator which allows the programmer to order a collection returned by a query in an arbitrary way. As previously, the operator should be much generalized in comparison to SQL, OQL and other existing query languages.

All the mentioned operators should have clear semantics, that is coherent for machine implementation and understandable for humans.

Req. 4 Extensibility and openness features

It should be possible to assure that BPQL may be easily extended. Its extension should be possible in the area of new language constructs, such as transitive closures, functions, triggers and views, built-in operators or functions, new features of workflow objects (e.g. dynamic object roles or encapsulation), their attributes or associations between them. At this stage of developing workflow metamodels it is impossible to assure that the kinds of workflow object attributes or features of objects will not change.

It should also be possible to extend BPQL of new application specific functions, procedures or views that would simplify query definitions. This functionality is crucial when either there are some data are out of the repository on which BPQL operates or a BPQL query uses a complex algorithm that needs to be implemented as a programming function. Both cases may occur in WfM systems. For instance a WfM system needs some particular application data, and these data may be stored in an application repository which does not provide BPQL interfaces or calculate some data 'on-the-fly' using complex heuristic algorithms.

Req. 5 Ready for optimisation

BPQL queries should be efficient. One of the main methods to achieve it is to support various optimisation techniques such as query rewriting, indexing, and query result caching. Optimisation requires very clean formal semantics, no exceptions, anomalies or special cases, and supporting some basic principles of query languages such as orthogonality and compositionality.

Req. 6 Adequate workflow mechanisms

Although in principle a query language is not proprietary to an application domain, there are essential, mentioned previously reasons to construct BPQL as biased towards WfMS. BPQL should offer appropriate mechanisms to operate on the workflow metamodel. These mechanisms should be able to extract information on individual workflow objects (e.g. activity, process instance) as well as associations between them. All: one-to-one, one-to-many (process definition - process instantiations), many-to-one and many-to-many associations (e.g. predecessor-successor relation between activities) should be supported. It should also be possible to extract in one query information about several associated objects of different object types (e.g. process definition, process instance and process instance states).

Nowadays, bigger and bigger number of process designers are non-programmers. They are usually familiar with UML diagrams but not with advanced definitions of various query languages. Therefore BPQL should address data structures that are as close as possible to the UML representation of the workflow process metamodel. Since this metamodel operates on objects and associations between them, it is very likely that BPQL should be a kind of an object-oriented query language.

Req. 7 Support for workflow monitoring functions

BPQL should also provide set of workflow monitoring functions. So far three types of such functions are considered:

- Functions to simplify operations on process control flow These operations includes finding the start and end activities, and determining predecessors or successors, direct or all, for one or more activities. The process flow functions are provided for process and process instance levels.
- Functions to assign workflow participants. They are used for 'better' selection of workflow participants to perform activities. There are various criteria which define what 'better' means. The most popular techniques are based on calculation of the number of the assigned tasks, their overall cost, and total time required to perform them. The other, most advanced ones, may be connected with risk calculation or time constraints prediction.
- Functions to calculate the Quality of Service (QoS) factors. These functions are able to provide information about the minimal, average and maximal values for individual QoS factors for each process definition.

The set of these functions should not be fixed. It should be possible to build new functions that may appear after detailed analysis of workflow applications.

Req. 8 Appropriate development status

In order to apply BPQL in a large scale it needs to be stable, mature and well documented. These features greatly support popularity and applicability of a language.

3.2 Workflow Management

The fundamental question is whether BPQL should initiate a new framework for workflow management. It should be noted that such workflow management frameworks as those from Workflow Management Coalition (more than ten standards for various workflow management topics) and Business Process Management Initiative (and its incarnations from IBM) have been continuously developed for almost ten years and represent effort of many research centres as well as leading workflow management companies and organisations. These frameworks have been

implemented (partially or fully) in many WfM systems, and these systems have been applied in many IT systems all over the world.

Obviously, we should take evolutionary rather than revolutionary approach. According to the main goal, this thesis has to define a Business Process Query Language on the basic of existing workflow management frameworks and new requirements for WfM systems. The main goal of the language concerns the possibility to define workflow monitoring capabilities. The language should extend (neither substitute nor modify) existing workflow process definition languages in order to make them more flexible and appropriate to express business processes. Such approach holds promise that BPQL will use existing knowledge about workflow management and may be widely applied in various WfM systems that are compliant with the mentioned frameworks.

Nowadays, there are several standard process definition languages such as XPDL, BPML or more web-oriented languages such as BPEL4WS and WSDL. Currently it seems that XPDL is the most mature and complete process definition language and therefore this language is considered in this thesis to be extended with BPQL. It should be noted, however, that other mentioned languages might also be extended with BPQL.

3.3 Standard Query Languages

The second fundamental question is whether BPQL should be developed from scratch or it should be a clone of a standard, well known query language such as XQuery, SQL or ODMG OQL. The justified reason to develop BPQL from scratch is that there is no popular or standard query language that could meet the mentioned requirements for BPQL.

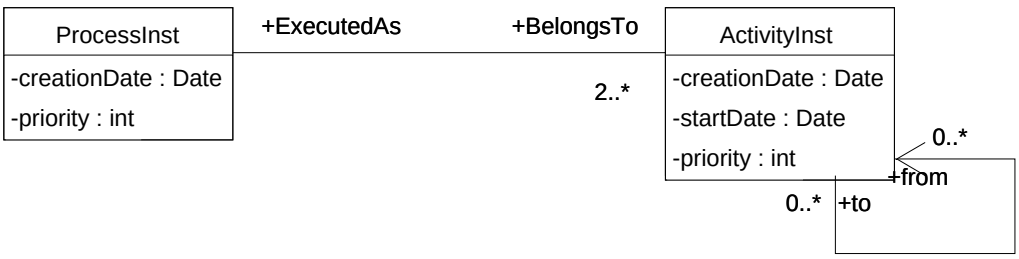

Figure 3.1 An example – a simplified metamodel

The discussion of pros & cons of the most known query languages is presented below. To compare features of the discussed languages we use a simple example which shows how basic elements of the metamodel as well as dependencies among them may be expressed in a given query language. It uses a part of the metamodel showing association among process instance and its activity instances. A UML model and sample data for this example are shown in Figure 3.1 and Figure 3.2.

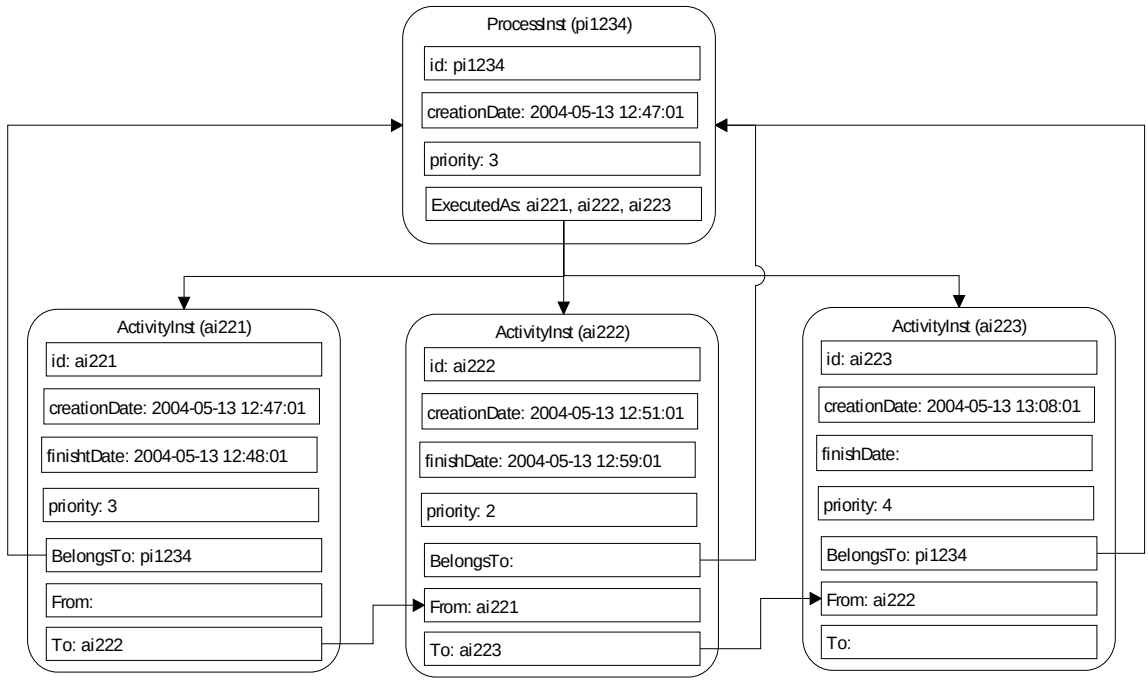

Figure 3.2 An example - a sample of data

3.3.1 XML and XQuery

Because of XML popularity the family of XML query languages comes as the first candidate for the mentioned standard language. XML is a simple but quite flexible text format. Every piece of data in XML is identified by tags (i.e. identifiers enclosed in angle brackets, like this: <...>). Compound data are represented by a hierarchical structure of nested data (and tags). For the example presented in Figure 3.1 one of the possible XML representations would look like:

```
<ProcessInst>
    <id> pi1234</id>
    <creationDate>2004-05-13 12:47:01</creationDate>
    <priority>3</priority>
    <ActivityInstList>
      <ActivityInst>
        <id> ai221</id>
        <creationDate>2004-05-13 12:47:01</creationDate>
        <finishDate>2004-05-13 12:48:01</finishDate>
        <priority>3</priority>
        <ToList>
          <To>ai222</To>
        </ToList>
      </ActivityInst>
      <ActivityInst>
```

```
                    <id> ai222</id>
                    <creationDate>2004-05-13 12:51:01</creationDate>
                    <finishDate>2004-05-13 12:59:01</finishDate>
                    <priority>2</priority>
                    <FromList>
                      <From>ai221</From>
                    </FromList>
                    <ToList>
                      <To>ai223</To>
                    </ToList>
                </ActivityInst>
                <ActivityInst>
                    <id> ai223</id>
                    <creationDate>2004-05-13 13:08:01</creationDate>
                    <priority>4</priority>
                    <FromList>
                      <From>ai222</From>
                    </FromList>
                </ActivityInst>
            <ActivityInstList>
        <ProcessInst>
```

Figure 3.3 An XML representation of the example

XML has found many applications. Original, and probably the most successful application of XML is to use it as a data interchange format. XML has been used in many data interchange standards and languages, also for workflow process definition. Beside this, in the era of Web-centric applications, XML is used for representation of so-called 'web-data'. For this usage some additional languages and tools to support XML have been developed, such as XML Style sheets Transformation Language (XSLT) [Clark99a]. Despite all marketing efforts, this way of using XML seems to come to a standstill. The reason is related to problems with XML structure efficiency and flexibility. Finally, because of the number, size and character of data stored in XML, this format has been used to store and manage data. Most of the famous vendors of data base management systems (DBMSs) either developed a native XML DBMS (e.g. Tamino) or extended their relational DBMSs of XML features (e.g. Oracle and Microsoft SQL Server). Unfortunately, due to the nature of XML, these DBMS-s represent only hierarchical data structures with no cycles, many-to-many relationships and other object-oriented features such as classes, methods, inheritance and polymorphism. Many-to-many relationships are to be represented as a hierarchical structures (one-to-many relationships) with additional references pointed to appropriate objects all over the XML document. For instance, in the example given in Figure 3.3 the data is represented as

a hierarchical structure. A process instance is on the top of this structure and its activity instances are represented as its nested objects. The relation between activity instances (i.e. *From-To* relation) is represented indirectly as references within individual activity instances (i.e. *From* and *To* tags).

Following this trend, a family of XML query languages have been proposed, such as XPath [Clark99b], XLink, XQL [Robbie1998], XML-QL [Deutch98], XIRQL [Fuhr00], QUILT [Robbie00] and recently XQuery [XQuery03]). From this family XQuery seems to be the most comprehensive and representative XML query language. In addition, it includes concepts and ideas developed in earlier XML query languages. However, its specification is still in the W3C working draft stage. Detailed discussion how BPQL requirements are satisfied by XML query languages – especially by XQuery - is presented below.

Req. 1 Clear, simple and expressive syntax

XPath and XLink have quite clear and simple syntax, however this concerns simple queries. For complex queries (e.g. nested subqueries) they are not expressive enough. On the other hand XQuery is expressive but quite complicated and unclear. There are 7 types of expressions in XQuery: path expressions (known from XLink and XPath), element constructors, for, let, where and return expressions (FLWR expressions), expressions involving operators and functions, conditional expressions, quantified expressions and expressions that test or modify data types. Some constructs seems to be artificial as FLWR expressions which are the analogue of the SELECT-FROM-WHERE construction known from SQL.

Considering the workflow metamodel, XQuery is able to represent all workflow objects and their attributes. However, there is a fundamental disadvantage of XML languages: as the consequence of the hierarchical representation of XML, these query languages are suitable for queries on the relation directly represented as the hierarchy and have difficulty to represent (complexity, readability, efficiency) queries that concern other associations. For instance, in the example given in Figure 3.3 the query to retrieve creation date of all the activity instances of the process instance with identifier *pi1234* may be represented in *XQuery*(path expression) in the following way:

```
/ProcessInst[@Id='pi1234']/ActivityInstList/ActivityInst/@creationDate
```

However, a query to ask about the creation date of the successor of the activity instance that is the start one has to be done in two steps (XQuery, path expression):

```
# Step1: retrieve identifier of the first activity -> firstActId
/ProcessInst[@Id='pi1234']/ActivityInstList/ActivityInst[FromList=empty]/@id
# Step 2: retrieve creationDate of the successor
/ProcessInst[@Id='pi1234']/ActivityInstList/ActivityInst/FromList[From=ActInstId]/@creationDate
```

From the practical point of view it may be observed that the greater number of relations implies the lower efficiency of XML DBMSs and their query languages.

Req. 2 Unambiguous, coherent and complete semantics

Semantics seems to be the "Achilles heel" of XML query languages. Despite attempts to define unambiguous, coherent and comprehensive semantics for these languages, still it is unclear, with many redundancy and 'for special use' cases. This is especially true for XQuery which is full of attempts to equip the language with constructs to do everything but without algorithmically defined how to do. Note that lack of coherent and complete semantics is the reason of different implementations and much undermines development of query optimisation methods.

Req. 3 Pragmatic universality, including a rich set of basic operators and built-in functions

XQuery is equipped with rich set of various operators and built-in functions. It supports arithmetic operators, comparison operators, logical operators and sequence-related operators. XQuery also supports conditional expressions. From the list of non-algebraic basic operators (in SBA terms) the following ones are supported: selection, projection and navigation, quantifiers (both universal – EVERY, and existential – SOME). FLWR expressions also introduce the dependent join operator but with limitations.

Req. 4 Extensibility and openness features

In every XML language adding a new type of classes to the metamodel or any new attribute (both simple or compound ones) is straightforward. In addition, as one of the new features, XQuery provides a mechanism for specifying user defined functions. Because of the ill-defined semantics the possibility of language's extensions is reduced. It is unclear how new language features will work in combination with already existing features.

Req. 5 Efficient and ready-for-optimisation language constructs

Once again, because of the hierarchical nature of XML structure, queries written in XML-oriented query languages that operate on many relations, especially many-to-many ones, are very inefficient. In addition, they can not to be optimised efficiently, because of unclear and incomplete semantics of all the XML languages (see Req. 2).

Req. 6 Adequate mechanisms to operate on the workflow metamodel

In general, XML languages provide sufficient mechanisms to operate on workflow entities and their attributes. However, as was stated earlier, XML languages are not prepared to cope with queries that use many relationships, especially many-to-many ones. Such queries are very inefficient and more complicated than they should be. In addition, since XML data storage model for workflow processes differs from the workflow metamodel, process designers need to map the metamodel to XML-like data structures; then to use XQuery. After such a mapping the metamodel

becomes more complex and less understandable than the original conceptual metamodel. For non-programmers it may be too difficult.

Req. 7 Support for workflow monitoring functions

Since XQuery provides a mechanism for specifying user defined functions, it is possible to define and then extend in the future the set of workflow monitoring functions that may be used in XML queries to simplify them. However, it should be noted that these user functions have to operate and return only XML tags, which differ from workflow objects. This probably implies higher effort to design, implement and maintain these functions.

Req. 8 Appropriate development status

XQuery documentation is prepared properly. So far there are several software vendors declaring that they implemented XQuery in their products (e.g. Tamino of Software A.G.).

3.3.2 SQL and SQL-99

The next candidate is the family of Structured Query Languages (SQL). These languages are very popular and widely used for relational databases. It is considered by many professionals as an essential factor of the commercial success of relational databases. There are several different standards of SQL, in particular SQL-86 (the most simple one), SQL-89 (reasonably complex and implemented in majority of relational DBMS), SQL-92 (very complex and entirely not implemented in any relational DBMS), SQL-99 [Melton01] (aka SQL3, extremely huge, eclectic and complex, not implemented entirely in any system, probably not implementable in full) and future standard SQL-2003 (known also as SQL-200n, which extends SQL-99). SQL-99 is claimed to be a superset of SQL-92. Because no DBMS vendor has decided to implement SQL-99 entirely, to encourage them the standardisation bodies (ANSI and ISO) have proposed some subset of SQL-99 known as Core SQL-99. It includes almost all features of SQL-92 and some small subset of the rest of SQL-99.

Decision of using some variant of SQL requires mapping the metamodel to relational datamodel or (in case of SQL-99) to object-relational model. Such a mapping makes the mismatch between the original conceptual metamodel and the targeted metamodel even more disadvantageous than mapping to XML-oriented datamodel.

In this thesis we make little attempts to address object-relational DBMS and SQL-99. Although object-relational DBMS as advertised as a “new great wave” in database technologies, common practice does not follow wishes and folders of big relational DBMS vendors. In the famous interview [Winslet02] David Maier claims that object-oriented extensions of relational DBMS are practically not used by database programmers. The probable reasons are the following: they are poorly integrated with SQL, with API-s such as ODBC and JDBC, and with the other

basic mechanisms of relational DBMS-s. As we have mentioned before, the standard of object-relational query language SQL-99 will likely be unsuccessful. Hence our comparison below concerns purely relational databases and SQL rather than their object-oriented extensions.

In a relational database data are represented by relations, where some of them represent real-world entities, and other represent relationships between entities. A relationship is represented either directly as an additional attribute of one of the related entities or as a new relation. In the latter case, the relation has to include at least foreign keys that represent references to the entities that participate in the relationship. All possible types of relationships are available: one-to-one, one-to-many, many-to-one and many-to-many. In Figure 3.1 one of possible relational representations of a part of our metamodel is presented in the 3rd normal form:

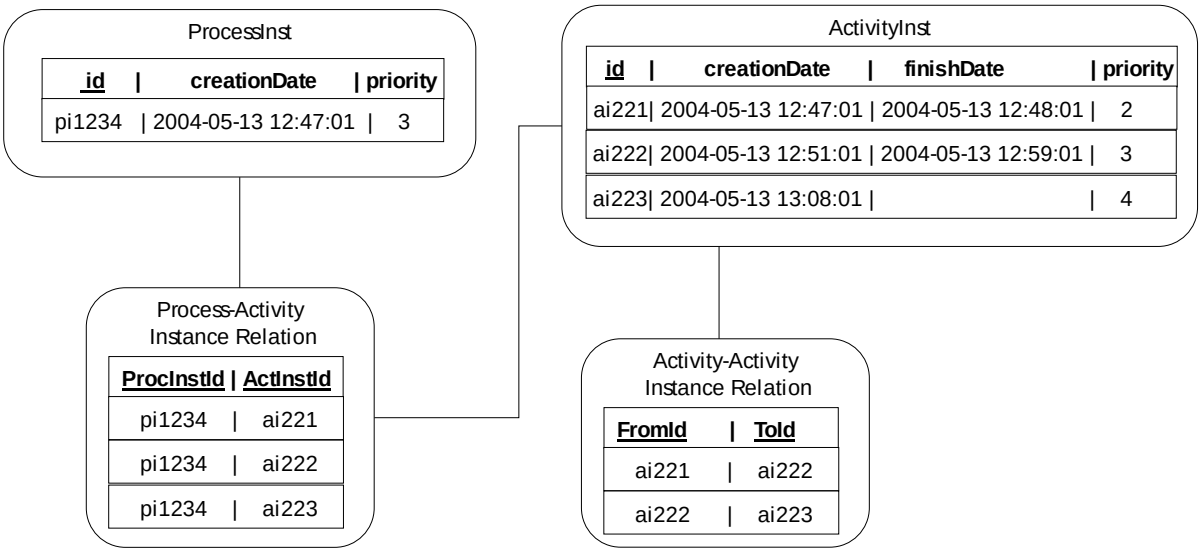

Figure 3.4 A relational representation of a part of the metamodel

Relational databases are widely used in many IT systems. From the very beginning SQL was the primary language to query this type of databases. It turned out to be useful and efficient in many various large-scale transactional systems. It provides various application programming interfaces, such as ODBC and JDBC. SQL has been extended in many ways - through database updating clauses, database views, stored procedures, triggers and in direction of programming languages (e.g. PL/SQL from Oracle). SQL was a subject of several standards, in particular SQL-89, SQL-92 and recently SQL-99. The detailed discussion how BPQL requirements are satisfied by the SQL family of query languages – especially by SQL-92 - is presented below.

Req. 1 Clear, simple and expressive syntax

Originally SQL had simple and quite clear syntax (SQL-86, SQL-89), however new specifications (SQL-92 and SQL-99) introduced many new and pretty complex extensions (e.g. object-relational features). The size of the consecutive SQL specifications is very characteristic: SQL-86 – about

100 pages, SQL-89 – about 120 pages, SQL-92 – about 600 pages and finally SQL-99 – about 2200 pages. In many Relational Data Base Management Systems (RDBMSs) only selected elements of the SQL-92 specification were implemented. Till now there is no evidence that SQL-99 has been implemented by any of RDBMS vendors. So far, few RDBMS vendors implemented only a selected small subset of the whole standard. The SQL syntax, although expressive, is frequently criticised for non-orthogonality, very big syntactic patters and unclear syntactic limitations. This caused the effect that professionals refer to as "SQL puzzles" [Celko97], i.e. some rather complex tasks are extremely difficult to express in SQL.

Req. 2 Unambiguous, coherent and complete semantics

The SQL specifications do not provide a complete semantics. There are opinions that relational algebra covers only several percent language constructs available in SQL [Subieta04]. Despite several attempts SQL has no formal semantics. Moreover, SQL has many semantics reefs, the most known are problems with *null values* [Date98] and restrictions on using the *group by* operator. For instance, if two relation attributes: A and B may have null values then the two following queries:

```
select sum(A)+sum(B) from R
```

and

```
select sum(A+B) from R
```

may return different results. Although the basic semantics of the *group by* operator is simple, in connection with more complex language constructs there are some semantic reefs (e.g. *null* value together with *group by* operator). Despite attempts to remove these semantics flaws in subsequent SQL standards, they still exists, and what is more, they have an amplified effect in further SQL specifications since the number of language constructs has been drastically increased.

Req. 3 Pragmatic universality, including rich set of basic operators and built-in functions

SQL is equipped with very rich set of various operators and built-in functions. It supports arithmetic operators, comparison operators, logical operators and sequence-related operators. In new specifications SQL also supports conditional expressions. From the list of non-algebraic basic operators SQL supports selection, projection and sorting. The navigation operator is impossible in relational data structures. SQL supports joins (via Cartesian product and selection) but not dependent join (it is proposed in SQL-99). Also quantifiers (both universal and existential) are not supported in SQL-89 and SQL-92, but the programmer can use other options (e.g. the exists operator), not so powerful as quantifiers.

Req. 4 Extensibility and openness features

Originally, standards SQL-89 and SQL-92 do not provide procedures, but various implementations of SQL extend it in this direction, e.g. Oracle PL/SQL extends it by stored procedures, triggers and

rules. In some implementations it is also possible to call a programming function written in Java or C# languages, however, with limitations concerning parameter passing. Extensions of SQL functionality is rather impossible because no programmer from outside of a vendor company has access to source code. SQL makes it easy to add a new table to the database and to add a new attribute to an existing table.

Req. 5 Ready for optimisation

For the last two decades SQL has proven its efficiency in many IT systems. However, because of problems with SQL semantics, optimisation of SQL queries (e.g. with null values and group by operator) still largely depends on individual implementations and is based mostly on heuristic algorithms. For simple queries not involving advanced options SQL optimizers are very efficient. However, there are examples of relatively complex queries when SQL optimisers give up. Such cases must be supported by manual optimisation or tips determined by programmers, based on "golden rules" of well-formulated queries (see Oracle).

Req. 6 Adequate workflow mechanisms

In general, all SQL specifications provide sufficient mechanisms to operate on the workflow process metamodel, assuming it will be mapped to the relational data structures. Unfortunately, the relational representation of the workflow objects and relationships among them is much different from an UML Class Diagram. This fact may limit drastically the number of users who will benefit from BPQL, since they would also need to know some SQL dialect (e.g. SQL-89) very well. For non-programmers SQL is a very complex programming language. A more reasonable solution from this point of view is to leave an UML class diagram as a metamodel schema, not converting it to normalized relations, and then, to use some very friendly object-oriented query language.

Req. 7 Support for workflow monitoring functions

As stated earlier, implementations of SQL e.g. in Oracle and MS SQL Server provide mechanisms (less or more advanced) for specifying user defined functions. Owing to this, it is possible to define and then extend in the future the set of workflow monitoring functions that may be used in SQL queries to simplify them. However, it should be noted that these functions have to operate and return only relational data, which differ from workflow objects. This probably would result in higher effort to design, implement and maintain workflow applications.

Req. 8 Appropriate development status

SQL-89 and SQL-92 have been (partially) implemented by many DBMS vendors and used by many IT systems. Its documentation is of very good quality. Despite some attempts, still there is no complete implementation of SQL-99.

3.3.3 Object- Oriented Query Languages - OQL and SBQL

The last alternative is the family of object-oriented database management systems and their query languages. There are many hopes and expectations connected with object-oriented DBMS. Object-oriented databases are able to represent classes and associations between them without any additional transformations, modifications or artificial extensions of the conceptual data schema expressed in object definition languages having the flavour of UML class diagrams. Owing to this fundamental feature, operating on data as objects and associations should be simpler and more natural. Also a corresponding query language should be easier to understand and use. Using the concept of objects should also reduce the impedance mismatch between query and programming languages. Moreover, object-oriented approaches introduce such features as inheritance, polymorphism and encapsulation. For the example presented in Figure 3.1 a possible object-oriented SBQL representation is depicted in Figure 3.2.

Despite of some effort focused on specification, design and developing object-oriented query languages, ODMG OQL [ODMG00] specification has many flaws [Alagic97, Subieta97]. It also lacks of formal semantics. So far there is no exact OQL implementation due to the fact that it addresses a specific ODMG object model, which is not exactly the same as data models of object-oriented DBMS, such as Gemstone, Object Store, Objectivity/DB and Versant. Optimisation methods for OQL seem to be not sufficiently matured. ODMG OQL [ODMG00] proposes a'la SQL language syntax which in most cases introduces only additional complexity and decreases its legibility and simplicity.

Fortunately, there is one more candidate that has recently joined the group, that is, the Stack-Based Query Language (SBQL) [Subieta95a, Subieta95b, Subieta04]. Unlike other presented candidates this language has simple, formal and coherent semantics. SBQL has very powerful query optimisation methods, in particular, methods based on query rewriting, methods based on indices and methods based on removing dead queries. Further methods are now under investigation. Currently SBQL has several implementations, in particular, for the ICONS European project [ICONS02], for XML repositories based on the DOM model, for the object-oriented DBMS Objectivity/DB and recently for the object-oriented server ODRA (Object Database for Rapid Application development). SBQL syntax is fully orthogonal and very simple thus seems to be easier to understood and use.

In this thesis we do not discuss ODMG OQL because there are serious doubts if the proposal is enough consistent and coherent for the entire implementation. We consider SBQL as

the best representative of object query languages, thus it is discussed in detail in the next sub-sections.

Req. 1 Clear, simple and expressive syntax

SBQL proposes very clear and simple syntax. For instance, to extract all the activity instances which were performed by *johnb* the following query may be formulated (see the database schema view presented in Figure 2.9):

```
ActivityInst where ( currState.opSuperState = 'Finished'  and  PerformedBy.Performer.name = 'johnb')
```

This query does not include any additional key word such as *select* or *from*, which in SBQL are not introduced. In our opinion these redundant lexical tokens are confusing and annoying for complex nested queries. The syntax of SBQL is very expressive. It is able to identify any workflow objects, their attributes, any of the defined associations, constants (also dictionary elements), numbers and texts. In fact, the SBQL syntax makes no constraint on tasks that can be accomplished through SBQL.

Req. 2 Unambiguous, coherent and complete semantics

SBQL semantics is based on the operational method (abstract implementation machine). The fundamental concepts of SBQL are taken from the stack-based approach (SBA) to query languages. In SBA a query language is considered as a special kind of a programming language. Thus, the semantics of queries is based on mechanisms well known from programming languages like the environment (call) stack. SBA extends this concept for the case of query operators, such as selection, projection/navigation, join, quantifiers and others. Using SBA one is able to determine precisely the operational semantics of query languages, including relationships with object-oriented concepts, embedding queries into imperative constructs, and embedding queries into programming abstractions: procedures, functional procedures, views, methods, modules, etc. Till now there is no another theoretical framework that combines universality of a data model, universality of a query language and precision of specification of semantics.

SBA is defined for a general object store model. Because various object models introduce a lot of incompatible notions, SBA assumes some family of object store models which are enumerated M0, M1, M2 and M3. The simplest is M0, which covers relational, nested-relational and XML-oriented databases. M0 assumes hierarchical objects with no limitations concerning nesting of objects and collections. M0 covers also binary links (relationships) between objects. Higher-level store models introduce classes and static inheritance (M1), object roles and dynamic inheritance (M2), and encapsulation (M3). For these models the formal query language SBQL is defined. SBQL is based on abstract syntax and full orthogonality of operators, hence it follows the mathematical flavour of the relational algebra and calculi. SBQL, together with imperative extensions and abstractions, has the computational power of programming languages. Concrete

syntax, special functionality, special features of a store model and a concrete metamodel allow one to make from SBQL a concrete query language, in particular BPQL.

Req. 3 Pragmatic universality, including rich set of basic operators and built-in functions

SBQL is equipped with rich set of various operators and built-in functions. It supports arithmetic operators, comparison operators, logical operators and sequence-related operators. From the list of non-algebraic basic operators all required ones are supported, that is selection, projection and navigation, dependent join, quantifiers (both universal and existential ones) and ordering.

Req. 4 Extensibility and openness features

SBQL provides mechanism to build user defined procedures and functions. Functions may return references to objects of simple types (i.e. integer, string, date) as well as more complex types (i.e. activity instance and process definition). In addition procedures and functions may have optional parameters. When a given procedure or function is called, default values are assigned to these parameters. SBQL can easily be extended to complex features, such as dynamic inheritance, encapsulation, updateable views, and others.

Req. 5 Ready for optimisation

Owing to the clear and coherent SBQL semantics, various query optimisation techniques are available, especially query rewriting (i.e. syntactic query transformation). Some of them generalise well known techniques known from relational databases while the others are originally designed for stack based approach [Plodzien00]. SBQL is also prepared for optimisations based on removing dead subqueries, for optimisation based on indices, for caching queries and for cost-based optimisation techniques.

Req. 6 Adequate workflow mechanisms

Mechanisms provided by SBQL seem to be both expressive and simply enough to operate on the workflow process metamodel. In SBQL it is possible to extract information about individual workflow objects (i.e. their attributes), and to analyse relations between them.

In addition, regarding the discussed languages, only in SBQL we are able to express queries which extract at the same time a list of objects and calculate for them some aggregated factors. For example the query:

Give a list of the process instances which are currently processed. For every process instance give a list of its activity instances, their total processing time (i.e. a sum of the processing time for every activity instance within the process instance), and the maximal their cost (i.e. maximal value of costs for individual activity instances within the process instance)

may be expressed in SBQL as follows (see the database schema view presented in Figure 2.9):

```
(
 ((ProcessInst where (currState.opSuperState = 'Running')) as  pi
join
 (pi.ConsistsOf.ActivityInst as ai)
)
join
 (sum(ai.duration) as aiDuration, max(ai.cost) as aiCost )
```

Since SBQL is an object-oriented query language, it operates on the same data storage model as defined in the workflow process metamodel. Such a feature enables the process designers to learn only the workflow metamodel which is the data storage model too. In addition, the size of the current SBQL specification can be estimated as 200-300 pages at most [Subieta04]. In fact, SBA and SBQL exist so far as a kind of theory, thus a commercial standard (in the spirit of ODMG or SQL-99) does not exists and is not anticipated.

Req. 7 Support for workflow monitoring functions

Since SBQL provides mechanisms for specifying user defined functions, it is possible to define and then extend in the future the set of workflow monitoring functions that may be used in SBQL queries to simplify them. Yet, these functions may operate and return references to workflow objects.

Req. 8 Appropriate development status

Since SBQL has been proposed recently, there are several its implementations only. The most complete is done for the EU project for Intelligent Content Management System (ICONS). There is one extensive book (in Polish) [Subieta04], more than 20 papers about SBA and SBQL (see e.g. http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/s/Subieta:Kazimierz.html), two doctoral theses, and several master theses done at the Polish-Japanese Institute of Information Technologies in Warsaw. Its complete documentation is under development. More advanced features of SBQL are still under research and development.

3.3.4 Final Comparison

The detailed discussion how some candidate query languages may satisfy the requirements for BPQL has been carried out in the previous sections. This discussion concerned three types of query languages, namely: XML-oriented, variants of SQL and object-oriented query languages, notably SBQL. For every language type, one the most appropriate, modern and mature query language has been selected. These three representatives were XQuery, SQL-92 and SBQL.

Despite many advantages, there are two fundamental disadvantages of using XQuery, and other XML query languages as BPQL foundation. First of all, these languages address hierarchical data structures with no cycles and many-to-many relationships. Unfortunately, many-to-many relationships are used quite often in the workflow process metamodel, for instance the predecessor-successor relation between activity instances is a many-to-many one. Secondly, so far, XQuery semantics is unclear, incomplete and partly inconsistent, thus coherent implementation and optimisation of XQuery can be problematic. Since XML data storage model differs from the UML-like conceptual workflow metamodel, process definers must understand precisely how particular concepts of the metamodel are mapped into XML structures and how these structures are to be queried in XQuery. For non-programmers it may be a too excessive assumption.

Another candidate query language for BPQL is SQL. Regarding commercial implementations SQL is mature, efficient and widely used. However, despite many advantages, there are serious disadvantages of using SQL. As in the case of XML query languages, SQL requires mapping the UML-like conceptual workflow metamodel into relational tables, thus initial concepts from the metamodel will be much deformed. According to experience, mapping UML class diagrams into relational structures results in approximately three times more concepts (relations, attributes, keys, dependencies between primary and foreign keys, etc.) and some original information is lost (e.g. information on cardinalities, information on inheritance, etc.). Querying such relational structures is also complex, especially for non-programmers. Again, according to experience queries in SBQL addressing an UML-like schema are approximately three times shorter than semantically equivalent SQL queries addressing a relational schema equivalent to the UML-like schema.

SQL presents also a big implementation challenge. So far, no vendor has implemented SQL-92 completely and there are only a few attempts to implement a small subset of SQL-99. One of the reasons is that SQL has no complete, clear and coherent semantics. Another reason is that designers of SQL take little care on orthogonality of syntax, they introduce many redundant constructs and there are little attempts to generalization some particular SQL constructs into universal ones. In effect, the specification of SQL-92 is huge and the specification of SQL-99 is extremely huge. Lack of well-defined semantics, non-orthogonality and many very particular language constructs cause that query optimisation is tricky rather than systematic and still much depend on heuristic algorithms and individual implementations. The size of SQL specification is also a big learning challenge for workflow designers. There are opinions that SQL, initially claimed to be a very simple query language for novices, in the SQL-99 case is the most complex programming language among all programming languages that were ever created by humanity. Summing up, despite commercial success and the object-relational manifesto promoting SQL as a future “intergalactic dataspeak”, for our goals SQL would be probably the worst solution.

From the first glance, object-oriented query languages seemed to be immature and offer no coherent and complete semantics. However, SBQL has shown that the object-oriented approach to query languages may offer simple, efficient, and the most appropriate foundation for BPQL. It offers UML-like datamodel, simple syntax and complete semantics. Consequently it provides strong mechanisms for query optimisation. In addition, SBQL operates directly on the workflow process metamodel which is easier to learn and use for process designers. Some disadvantages concerns only maturity and stability - SBQL has been proposed recently and so far there are only few implementations. An integrated summary of the discussed query languages is presented below.

Table 3.1 Comparison of the standard query languages

\ Core query language *BPQL requirement *	*XML* *XQuery*	*SQL* *SQL-92*	*OOQL* *SBQL*
1. Clear, simple and expressive syntax	***medium***	***medium***	***high***
2. Unambiguous, coherent and complete semantics	***low***	***low***	***high***
3. Pragmatic universality, including rich set of basic operators and built-in functions	***high***	***high***	***high***
4. Extensibility and openness features	***high***	***high***	***high***
5. Ready-for-optimisation	***low***	***low***	***high***
6. Adequate workflow mechanisms	***low***	***medium***	***high***
7. Support for workflow monitoring functions	***high***	***medium***	***high***
8. Appropriate development status	***high***	***high***	***medium***
9. Retaining an original UML-like workflow metamodel	***medium***	***low***	***high***
Total – how BPQL requirements are satisfied	***low***	***low***	***high***

This comparison shows that the Stack Based Query Language is most promising to satisfy the BPQL requirements. What may be done better for SBQL at this stage of its development, is putting more stress on its popularisation in both research groups as well as among DBMS software vendors. In the next chapter we present BPQL more in detail, focusing on its syntax and semantics.

4 Business Process Query Language Framework

On the basic of the workflow process metamodel and BPQL foundations described in the previous chapters, now it is possible to define the framework for BPQL. This framework describes the three major aspects of BPQL: its syntax, semantics and pragmatics. Syntax is defined as a set of rules and presented formally in the Extended Bachus-Naur Form (EBNF, [ISO14977]) in Appendix A. Semantics is given by description of the query evaluation mechanism and definition of the query evaluation procedure for every BPQL operator and construct. Finally, pragmatics is given by practical examples of BPQL queries.

4.1 Syntax

Syntax determines how BPQL queries are constructed. First we describe what are the BPQL tokens and how they are categorised. A *BPQL token* is an atomic character string that may be included in a BPQL query. Then we define a set of BPQL syntactic rules.

4.1.1 Language Tokens

In the BPQL grammar we will use the following sets of tokens:

- L – set of literals. A *literal* may be numeric (i.e. an integer or float point number), Boolean (i.e. true and false) or alpha-numeric (i.e. a string of characters surrounded by apostrophes). Some of literals may have additional meaning such as alpha-numeric literals which are considered as dates.
- N – set of *names* specified for entities defined within the workflow process metamodel (e.g. attribute names, association names, class names). The content of this set is common for all BPQL queries which operate on the same metamodel and the same list of available functions and procedures. The set must also includes temporary ad hoc names that the users can use in queries to name their parts.
- O – set of BPQL operator names, imperative construct names, function and procedure names (both built-in and user-defined) and other reserved words (e.g. where, as). The content of this set is common for all BPQL queries and all metamodels.
- Parentheses and other tokens to determine syntactic structures of queries.

4.1.2 Rules

On the basic of the above sets we define the set of rules for BPQL queries. The meaning of the operators and functions used in these rules is defined in the next sections.

R1. Any element of *L* is a query. For instance: 2, 3.14, 'alaBama123', '2004-05-07', true belong to *L*.

R2. Any element of *N* is a query. For example: *ProcessInst* (class), *currState* (attribute), and *InstantiatedAs* (association) belong to *N*.

R3. If Δ is an algebraic unary operator and *q* is a query then Δ*(q)* is a query. Examples of such operators are: *sqrt*, *sum*, *avg*, *log*, -, *not*. For selected operators (e.g. -, *not*) the brackets may be omitted.

R4. If Δ is an algebraic binary operator and q_1, q_2 are queries then $q_1 \Delta q_2$ is a query. Examples of such operators are: =, +, -, *, /, <. >, *and, or*.

R5. If θ is a non-algebraic operator and q_1, q_2 are queries then $q_1 \theta q_2$ is a query. Examples of such operators are: selection (where), projection/ navigation (.), dependent join (join), quantifiers, etc. For quantifiers the following syntax is used: *for some* q_1(q_2) and *for all* q_1(q_2). This syntax is equivalent to the universal one: $q_1 \exists q_2$ and $q_1 \forall q_2$.

R6. If *q* is a query and n ∈ N then *q* as *n* is a query. The *as* operator is an unary algebraic operator parameterised by the name *n*. This operator will be used in situations where a temporal name (so called *synonym*) is needed.

R7. If *q* is a query and n ∈ N then *q* group as *n* is a query. The *group as* operator is an unary algebraic operator parameterised by the name *n*.

R8. If *q* is a query, then *(q)* is a query. This rule makes it possible to use brackets freely. Most of the brackets included in queries will be omitted according to the operator's priority rules and the evaluation rules.

R9. If *n* ∈ O and has *k* parameters and q_1, ..., q_k are queries then $n(q_1, ..., q_k)$ is a query.

R10. If *n* ∈ O and has no parameter then *n()* and *n* are queries. These two queries are equal to each other.

R11. If q_1, ..., q_k are queries then struct$(q_1, ..., q_k)$, bag$(q_1, ..., q_k)$, *and* sequence$(q_1, ..., q_k)$ are queries. For the *struct* operator it is possible to use its simplified form $(q_1, ..., q_k)$.

R12. If q_1, q_2 are queries then if q_1 then q_2 is a query.

R13. If q_1, q_2, q_3 are queries then if q_1 then q_2 else q_3 is a query.

4.1.3 Examples

This section presents some examples of queries compliant with the above BPQL syntax. This section does not explain yet the meaning of the queries. The examples use the workflow entities presented in the database schema views in Figure 2.4, Figure 2.5, and Figure 2.9.

Ex.1 Simple queries

a) text: *'lorien', 'alabama1956', 'poland_Polish'*

b) integer: *-123, 0, 1972* and float: *-123.45, 0.0, 0.4567*

c) Boolean: *true*

d) object name: *ProcessInst, Activity,*

e) object attribute name: *cost, priority*

f) association name: *InstantiatedAs, From*

Ex.2 Standard queries

a) Extract all activity instances created by the 1st of December, 1999:

```
ActivityInst where (creationDate < '1999-01-01')[2]
```

b) Extract a list of users (i.e. their names) which performed the activity instances returned by the query from the previous example:

```
distinct((ActivityInst where (creationDate < '1999-01-01')).PerformedBy.Performer.name)
```

c) Get the number of the performers returned by the previous query:

```
count(distinct((ActivityInst where (creationDate < '1999-01-01')). PerformedBy.Performer.name))
```

d) Get the number of activities which have to be performed by *johnb* (i.e. remain in the *Created* or *Running* superstates).

```
count(distinct(
        ActivityInst where (
                ((currState.opSuperState) in ('Created', 'Running')) and
                ((PerformedBy.Performer.name) = 'johnb')
        )
))
```

e) Get the number of activities which have to be performed by the user whose name is returned by the function *funX*. List of all possible performers is passed as the parameter of this function. Possible performers are defined as resources in the IT system:

```
count(distinct(
        ActivityInst where
                ((currState.opSuperState) in ('Created', 'Running')) and
                ((PerformedBy.Performer.name) = funX(Resource))
        )
))
```

f) Get a list of activity instances together with information about the process instance which this activity instance belongs to:

```
ActivityInst join (BelongsTo.ProcessInst)
```

[2] For this example and the other examples throughout this document, we assume an automatic conversion a string into a date.

Ex.3 Complex queries

a) Extract a list of activities for which the current delay is greater than the average delay for of the all currently processed and delayed activity instances.

```
((ActivityInst where (
        (currState.opSuperState) = 'Running' and
        deadline < CurrDate()
 )) group as ai).
(ai where (CurrDate()[3] - deadline) > avg(CurrDate() - deadline))
```

b) Calculate the total time required to finish all tasks currently processes by *johnb*.

```
sum(
 ((Resource where name = 'johnb').IsPerformer.Performer.Performs.ActivityInst
        where (currState.opSuperState) in ('Created','Running')
 ).(
        if (InstanceOf.Activity.duration > duration)
        then (InstanceOf.Activity.duration - duration)
        else 0
)
```

4.2 Semantics

Semantics answers the question what is the formal meaning of BPQL queries and why they need to be constructed is such a way. In order to understand semantics, first we present the architecture of the query evaluation mechanism and describe the concept of ENVironment Stack (ENVS) and Query REsult Stack (QRES). In the second part, we define the meaning of the individual BPQL operators (algebraic and non-algebraic) and imperative constructs.

4.2.1 The Overall Architecture of the Query Evaluation Mechanism

During execution of a BPQL query, the query evaluation mechanism operates on workflow objects (both volatile and persistent ones), uses local variables and calls appropriate procedures. Workflow objects are managed by a *data storage*. Local variables, function call parameters and other query processing elements (e.g. parameter name binding) are controlled by the *environmental stack*. The entities stored on this stack depend on the data storage content and current query results (also temporary ones). During query evaluation all the query results (temporary ones as well as the final one) are managed by the *query result stack*. Both stacks refer to the workflow objects by their references. Stack may also contain other elements, such as numerical values, string values, binders1 (named values), etc.

[3] CurrDate is a built in function to get the current date and time.

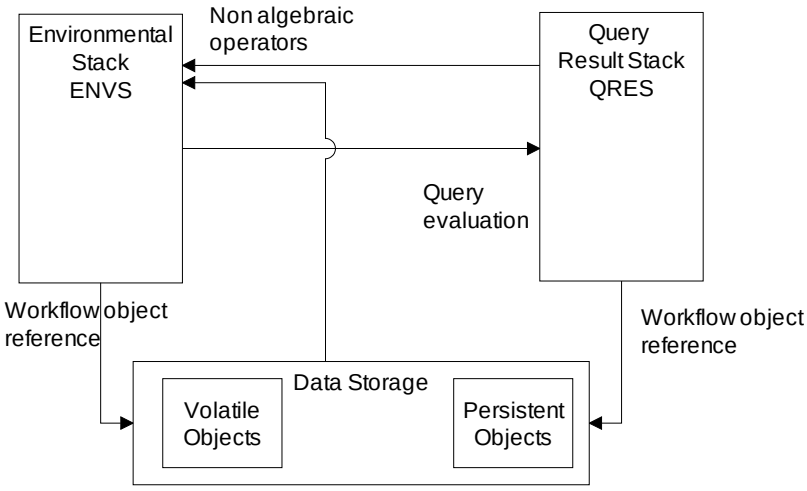

Figure 4.1 Architecture of the query evaluation mechanism

For query optimisation some additional elements of the presented architecture have to be introduced. In particular, optimisation by query rewriting introduces such new elements as static ENVS, static QRES and a meta-data storage. During query compilation these structures are used by the mechanism that rewrites a given query into another, semantically equivalent, but more efficient query.

4.2.2 Environmental Stack

The idea of ENVS is based on the environmental stack (call stack) concept well known and widely used in programming languages. Originally, the environmental stack is used to store all local programming entities that are necessary to execute a given procedure call (or function/method/operation call). Examples of such entities are local variables, local constants, and procedure parameters. The content of the call stack strictly depends on the operation context (local and global environment), that is, on the place in the program where a given programming command is executed.

The ENVS stack is adopted for BPQL semantics. It is responsible for:

- controlling the scope of names occurring in queries and their current bindings,
- storing local objects or variables (for procedures, functions or methods),
- storing procedure, function or method arguments (parameters),
- keeping the return trace for a procedure, function or method call.

ENVS follows the following rules:

- single objects and collections of objects stored in ENVS stack are treated in the same way,
- from the conceptual point of view, the maximal size of ENVS is unlimited,
- ENVS consists of *sections*. Every section includes information about the environment for a recognisable part of BPQL query, for example, an environment for a function call, or an environment of an object interior,

- the most local section (e.g. section of the procedure which is currently executed) will be placed on the top of the ENVS. Sections for the functions which were called earlier, are located in ENVS lower than the most local section,
- at the bottom of the stack global sections are located. They include common function libraries, global variables, references to database objects, etc.,
- ENVS treats persistent and volatile data in the same way,
- ENVS also stores information about queries definition (e.g. synonyms, quantifier variables),
- ENVS stores neither objects nor query results. The former information is stored in the data storage, while the latter information is managed by the query result stack QRES.

The basic entity in ENVS is a *binder*. A binder is a pair <*n, x*> (written as *n*(*x*)) where *n* is a name (e.g. object name, procedure name, variable name, attribute name, etc.) and *x* is a reference to the object having this name (at given time). The role of the *n*(*x*) binder is to bind the name *n* occurring in a query. The result of binding is *x*. For any name occurring in a query a corresponding binder should exists on ENVS. In a more general setting *x* is generalized to any query result.
ENVS consists of sections, which are ordered and managed according to the FIFO principle. Each section is a set of binders. Binding follows the "search from the top" rule: first the youngest section at the ENVS top is visited, then the section below the top, etc. up to the global sections on the ENVS bottom.

ENVS, as an abstract data type, provides three operations:

- *push* – add a new element at the top of the stack. The number of sections in the stack increases by one.
- *pop* – extract the element which is placed on the top of the stack. The number of sections in the stack decreases by one.
- *bind* - a function having a parameter which is a name. It searches the stack for binders having a particular name *n*, according to the "search from the top" rule and taking into account static scoping rules (which require omitting some stack sections). After finding a corresponding binder the search is terminated. All binders having name *n* that are found in the closest to the top section contribute to the final result of binding (in case of multiple bindings a bag of their *x* parts is returned).

Detailed description of the ENVS mechanism can be found in [Subieta04].

4.2.3 Query Result Stack

QRES is a generalisation of the arithmetic stack known from implementation of programming languages. In a classical case, an arithmetic expression is transformed to another equivalent

expression written in the Reverse Polish Notation (RPN). Then, this RPN expression is evaluated. During this evaluation the arithmetic stack is used to store arguments, temporal results and the final result. A simple example how arithmetic expressions may be evaluated is given below.

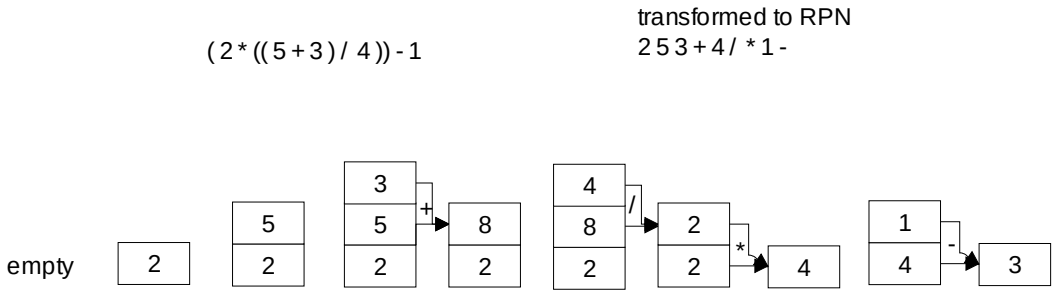

Figure 4.2 The arithmetic stack states for a simple expression

QRES differs from the above arithmetic stack in two aspects. Firstly, the stack elements are any temporal and final query results. When query evaluation starts, QRES has no element. After the query evaluation QRES should include the final result only. A query result may include one or more objects. If a BPQL function (or operator) is evaluated and it requires *n* arguments, then these arguments are taken one after another from the top of the QRES stack (i.e. *pop* operation on *n* stack elements). After evaluation, the function result is stored on the top of the stack (i.e. *push* operation).

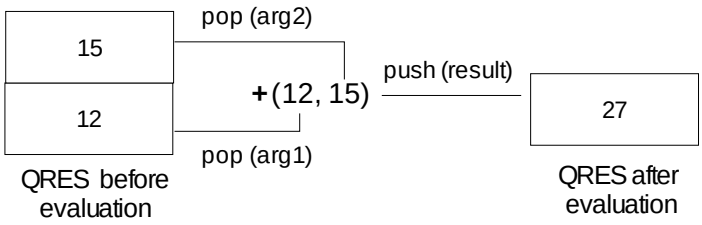

Figure 4.3 QRES stack - a sample of a query evaluation

Secondly, QRES may also include other elements required to evaluate queries, especially iteration counters operating on result collections. These elements are not described in this chapter.

QRES, as an abstract data type, provides four operations:

- *push* – add a new element at the top of the stack. The number of the elements in the stack increases by one.
- *pop* – extract the element which is placed on the top of the stack. The number of elements in the stack decreases by one.
- *top* – read the top of the stack,
- *empty* – check whether the stack is empty.

4.2.4 Query Evaluation Procedure

The query evaluation procedure (referred further to as *eval*) is a recursive procedure which takes as the argument a BPQL query, evaluates it and returns its results. The procedure is driven by a parser which generates a syntactic tree of the parsed query (i.e. subqueries, operators). A template of the *eval* procedure may be expressed in the pseudo-code as follows:

```
(1) procedure eval( q : BPQL query)
(2) begin
(3)       parse( q );  // syntactical tree generation
(4)       case q recognised as ... :
(5)       ...
(6)       case q recognised as  :
(7)       ...
(8) end;
```

During execution, the *eval* procedure operates on workflow objects stored in the data storage, ENVS stack and QRES stack. The eval execution result is stored finally on the top of the QRES stack. For BPQL queries without 'side-effects' (i.e. those which do not call methods or functions that change the state) the following assumptions for the *eval* procedure were made:

- It does not change the state of the data storage (i.e. it may only read workflow objects).
- During query evaluation, it may change the ENVS stack state. However, when this evaluation is finished, the ENVS state has to remain the same as it was before the evaluation.
- During query evaluation, it may not change the part of QRES stack that was present before the evaluation. It is possible to push and pop some elements during query evaluation, but after that QRES stack has to have only one more element comparing to those present before the evaluation.

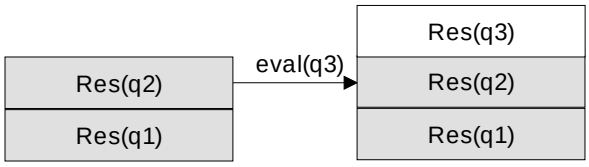

Figure 4.4 QRES stack and query evaluation

The further definition of BPQL semantics is driven by abstract syntax. That means the *eval* procedure is supported by a parser which is able to generate a syntactic tree of the query (i.e. subqueries, operators). An example of such syntactic tree for a simple query: *ProcessInst* ***where*** *cost* > 300 is presented below.

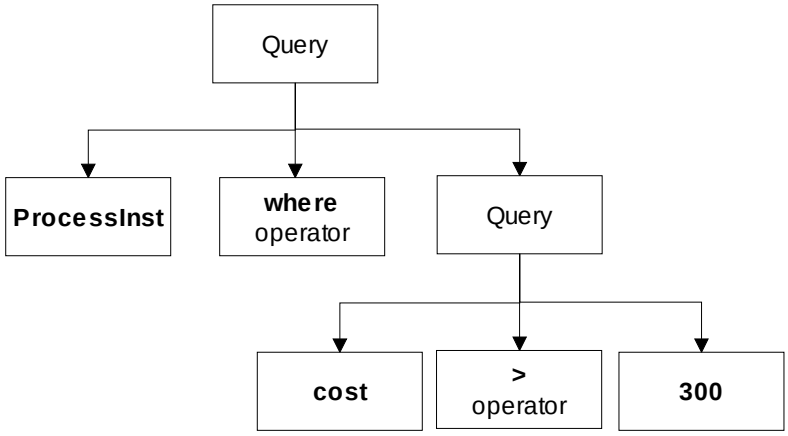

Figure 4.5 The syntactic tree for a simple query

The result of query evaluation is determined on the basic of the subqueries evaluation results and appropriate operators. The evaluation process is recursive, first the leaves of the syntactic tree are evaluated, then again evaluation results together with appropriate operators are evaluated. Such evaluation is repeated until the top of the syntactic tree is reached and the final query evaluation result is determined. This feature of BPQL syntax and semantics is called *modularity*.

The temporary results places in the QRES stack are consumed by BPQL operators. For example, the *greater than* operator (>) takes two elements from the QRES top (i.e. pop operation), adds them to each other and saves the result back on the QRES top (i.e. push operation).

4.2.5 Literals and Names

An elementary BPQL query is a lexical element ($l \in L$) or a name ($n \in N$). The *eval* procedure for such a query may be expressed in the pseudo-code as follows:

```
(1) procedure eval( q : BPQL query)
(2)  begin
(3)       parse( q );  // syntactical tree generation
(4)       case q recognised as l ∈ L :
(5)                 push( QRES, l );
(6)       case q recognised as  n ∈ N :
(7)                 push( QRES, bind( n ) );
(8)       case .....
          .....
(9)  end;
```

If the query is a literal (line 4), the *eval* procedure pushes on the QRES top this literal (line 5). If the query is a name (line 6), the *eval* procedure calls the *bind* function with this name *n* as a

parameter. The function accomplishes binding the name on the ENVS stack and then the binding result (usually references to objects named *n*) is pushed on the QRES top (line 7).

4.2.6 Algebraic Operators

One of the basic features of algebraic operators is that it does matter in which order their arguments are evaluated. Owing to this feature any algebraic operator operates only on objects stored in QRES and may be expressed in the *eval* procedure. The pseudo-code representation of the *eval* procedure is given below (Δ denotes an algebraic operator, and $\underline{\Delta}$ – function that represents this operator):

```
(1) procedure eval( q : BPQL query)
(2)  begin
(3)       ...
(4)       // algebraic unary operators
(5)       case q recognised as Δ (q1) or Δq1 (where q1 is a query and Δ is an unary algebraic operator):
(6)       begin
(7)         resQ1: Result;                  // local variable represents the result of q1 evaluation
(8)         eval(q1);              // call evaluation procedure for q1, its result will be stored on the QRES top
(9)         resQ1:= top(QRES); pop(QRES) // assign to resQ1 the result of the q1 evaluation
(10)        push(QRES, Δ(resQ1))       // store the result of the overall query evaluation on the QRES top
(11)       end;
(12)      ...
(13)       // algebraic binary operators
(14)      case q recognised as  q1 Δ q2 (where q1, q2 are queries and Δ is a binary algebraic operator):
(15)      begin
(16)        resQ1, resQ2: Result;          // local variables represent the result of q1 and q2 evaluation
(17)        eval(q1);              // call evaluation procedure for q1, its result will be stored on the QRES top
(18)        eval(q2);              // call evaluation procedure for q2, its result will be stored on the QRES top
(19)        resQ2:= top(QRES); pop(QRES) // assign to resQ2 the result of the q2 evaluation
(20)        resQ1:= top(QRES); pop(QRES) // assign to resQ1 the result of the q1 evaluation
(21)        push(QRES, Δ(resQ1, resQ2))    // store the result of the overall query evaluation on the QRES top
(22)      end;
(23)      .....
(24) end;
```

Similarly for algebraic operators having more than 2 arguments. Detailed description of the most important algebraic operators is included in the consecutive sub-sections.

4.2.6.1 Typical Operators and Functions

Typical operators and functions were categorised into functional groups. These groups are described below. Note that in a BPQL query arguments of operators and functions are also BPQL queries. This is the typical orthogonality principle, unfortunately, violated in many cases in SQL.

Comparison operators

There are two basic binary comparison operators: *equal to* ('=') and *unequal to* ('<>') operators. These operators are defined for all object and value types. The expression *x* = *y* returns *true* (a Boolean value) if *x* and *y* are semantically identical. Otherwise, it returns *false*. The expression *x* <> *y* returns *true* if *x* and *y* objects are semantically different (i.e. are not semantically identical). Otherwise, it returns *false*. For objects we assume shallow comparison, i.e. *x* = *y* returns true if x and y are the same reference. For values or atomic types such as integer or string the comparison returns true if their types and values are the same. For complex values (e.g. structures) the comparison returns true if the structures are isomorphic up to the order of structure fields (recursively). We assume that if types are different (e.g. *x* is integer and *y* is float) then automatic coercions (type change) are called. All coercions are resolved during parsing and added to syntactic trees as unary algebraic operators. The same concerns dereference operators which are also usually implicit and inserted into syntactic trees during parsing. Note however that originally in SBQL besides compile-time coercions there are also run-time coercions.

There are several other comparison operators: *less than* ('<'), *less than or equal to* (<=), *greater than* (>), *greater than or equal to* (>=). These operators are defined for value types where it is possible to determine the linear order. Examples of such types are numeric types (Integer, Float, etc.), text types (e.g. String), and date types. The meaning of these operators is strictly connected with individual types.

For the string type there is one additional operator: *like*, known from SQL The operator makes it possible to ask for string that matches a given template. For example, the condition: *x like '%town'* may be satisfied for *x* containing strings such as 'town', 'downtown', 'townhill'.

Arithmetic functions

There are several standard arithmetic operators: *multiplication* ('*'), *division* ('/'), *addition* (+), and *subtraction* (-). These operators are defined for numeric numerical values of types integer, long, float, double, etc., as usual.

Math functions

These functions operate on numerical values. The standard math functions are: round value

(*round*), truncate value (*trunc*), absolute value (*abs*), logarithm (*log*), square root (*sqrt*), power (*pow*), sine (*sin*), cosine (*cos*), and tangent (*tan*).

String functions

These functions operate on textual objects. The standard string functions are: left padding (*left*), right padding (*right*), concatenation (*concat*), string replacement (*replace*), extraction of a substring (*substring*), and trim operation (*trim*) . The above functions return strings. In addition, there is also a function *length* to evaluate the size of a string.

Collection functions

These functions operate on collections. The standard functions are: counting a collection of elements (*count*), checking whether a collection is not empty (*exists*), removing redundancy from the collection (*distinct*). The *count* function returns the number of the elements included in the collection (i.e. the Integer object) . If there is no element in the collection, the function returns 0. The function *exists* returns *true* (i.e. *Boolean* object) if there is at least one element in the collection. Otherwise it returns *false*. The distinct function returns a collection that is created from the original one after removing redundant elements.

Aggregate (numeric) functions

These functions operate on a group of numeric objects. The standard functions are: sum of numbers (*sum*), the average value of the set (*avg*), the minimal value in the set (*min*), the maximal value of the set (*max*) .

4.2.6.2 Type Coercions

A type coercion function converts the type of an object or change its representation. Such a function may be expressed implicitly, when it is required from the query context or explicitly, when it is included directly in the query. If a type coercion fails, the query compiler returns an error message or an exception. The typical type coercion functions and operators are listed here:

- usually **Dereference operators***(deref)* are expressed implicitly. They are called when the reference to an object needs to be converted into the object value. For instance, in the query:

```
ActivityInst where (creationDate < '1999-01-01')
```

'creationDate' returns the reference to a Date object that represents the *creationDate* attribute in an activity instance object. The dereference function is forced by the '<' operator which changes the reference to 'creationDate' into the value of the *creationDate* attribute. Sometimes the dereference function may be explicitly included in the query.

- **Type conversion functions** take an object of a given type A and returns another object of type B. The number of type conversion functions depends on the number of types available in BPQL. In many cases they are automatically added during query evaluation. For instance, if *x* is an Integer and y is a Float, then in the expression: *x+y* the *x* object will be automatically converter into a Float value.
- **Cast operators** *(cast)* change the type of the object reference. For instance, in the following query:

```
((CompoundActivity)(Activity where id = '7')).count(CalledWith.ParameterMapping)
```

  firstly the reference to the activity with the identifier 7 is extracted and then this reference is converted into another reference that points to a compound activity object. Finally the number of the parameters of the compound activity is returned.

  In comparison to C/C++ we assume that cast operators may only generalize (up-cast) or specialize given objects (down-cast), i.e. cast operators follow the inheritance hierarchy of classes.
- **Collection type conversion functions** are similar to the cast operators, but they change the type of a given collection, not the types of its elements. For instance, in the following query:

```
(sequence) ActivityInst where (creationDate < '1999-01-01')
```

  the activity instance objects returned as a bag are converted into the sequence collection (no special order required).
- **Element-to-collection and collection to element conversion functions** enable an element to be represented as a collection and vice versa. Usually this construction is used implicitly. For instance, if for the following query:

```
(ActivityInst where (creationDate < '1999-01-01')).(cost - 123)
```

  the bag of cost references is automatically converted into a single element (what is necessary for subtraction operator). If the conversion is impossible, an exception is raised.

4.2.6.3 Collection and Structure Operators

A collection is a group of objects or values that need to be processed together. The size of collection is not restricted and usually not predictable. Usually collections contain elements of the same type, although this is not always obligatory. In BPQL there are two types of collections: bags and sequences. Additionally, BPQL has the *struct* (structure) construct that generalizes the corresponding construct known from C/C++, Java, CORBA, ODMG, etc. A struct collects (named) values of different types. Usually the size and type of structures is known during compile time. The order of elements within struct is sometimes essential.

The basic collection type is a *bag*. All BPQL queries results are returned as bags (sometimes as one-element bags). A bag may contain duplicate elements. Two examples of a bag collection are given here:

- A bag represents a group of odd numbers less than 10.

```
bag(1, 3, 5, 7, 9)
```

- An 'implicit' bag represents a group of activity instances that have been performed by 'johnb'.

```
ActivityInst where ((PerformedBy.Performer.name) = 'johnb')
```

A *sequence* is an ordered set of elements of one type. In a sequence it is possible to retrieve its elements by their indices. A *struct* represents a structure of values of various types, however (in contrast to C/C++, etc.) it is not required to assign names to all of the structure elements.

There are five basic operators to operate on collections, namely: union (*union*), intersection (*intersect*), difference (*diff*), subset (*in*), equality (=). The meaning of the mentioned operators is similar as for those defined in the mathematical set theory. For instance, if we want to find all users that are either the owner of a process definition or have performed any activity instance, we may write the following query:

```
distinct((ActivityInst.PerformedBy.Performer.Represents.Resource.name) union (ProcessDef.owner.name))
```

4.2.6.4 Alias Operators

An *alias operator* may be considered as an additional name assigned to the result of a given query. For instance, the set of activity instances returned by the following query may be recognised by the additional name *actInst1*:

```
(ActivityInst  where (currState.opSuperState) = 'Finished') as actInst1
```

The labelled result of a query may also be used by another query on order to simplify its definition. It is especially useful in the case when the query result once evaluated are used many times within another parts of a compound query. For instance, the set of activity instances labelled by the alias *actInst1* may be used twice: to evaluate results of two subqueries: one to extract information about the finished activity instances which were performed by *johnb*, and one to extract information about the finished activity instances which were started before the 1st of January, 2004:

```
((ActivityInst  where (currState.opSuperState) = 'Finished') as actInst1).
(
        actInst1 where (PerformedBy.Performer.name) = 'johnb'
union
        actInst1 where startDate < '2004.01.01'
)
```

Following the concept described in [Subieta04] the alias operators will be defined as unary algebraic operators parameterised by a name (i.e. alias). There are two alias operators: *as* and *group as*. The *as* operator labels all values returned by a given query. If the *q* query returns a bag: **bag**{x_1, x_2, x_3, ...}, then the query: *q* ***as*** *n* (where n ∈ N) returns a collection of binders: **bag**{ $n(x_1)$, $n(x_2)$, $n(x_3)$, ...}. The alias operator does not change a collection kind. In particular, if q returns **sequence**{x_1, x_2, x_3, ...}, then *q* ***as*** *n* also returns **sequence**{ $n(x_1)$, $n(x_2)$, $n(x_3)$, ...}. The alias operator may be nested. For instance, if q returns a bag: **bag**{x_1, x_2, x_3, ...}, then the query: (*q* **as** n_1) **as** n_2 returns **bag**{ $n_2(n_1(x_1))$, $n_2(n_1(x_2))$, $n_2(n_1(x_3))$, ...}.

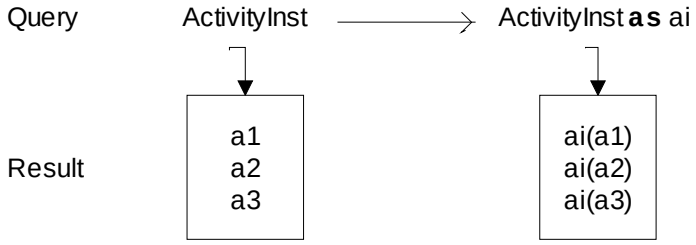

Figure 4.6 Application of the as operator

The *group as* labels the whole returned collection, not its individual elements (unlike for the *as* operator). If the query *q* returns **bag**{x_1, x_2, x_3, ...}, then the query *q* ***group as*** *n* returns a single binder to the whole collection: n(**bag**{x_1, x_2, x_3, ...}).

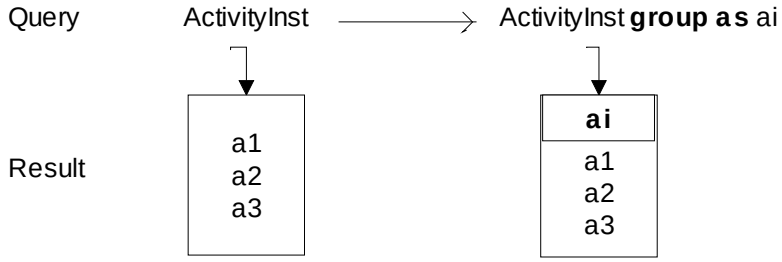

Figure 4.7 Application of the group as operator

4.2.7 Non-algebraic Operators

Non-algebraic operators are considered as the essence of the stack-based approach. All non-algebraic operators are binary. This group includes selection, projection/navigation, dependent join and quantifiers (existential and universal ones).

Following the discussion included in [Subieta04] it may be observed, that though the non-algebraic operators seem to be similar to algebraic ones, their semantics may not be expressed easily using algebras. Well defined and formal theories, such as the relational algebra, are able to cover only selective subsets of functionality provided by SQL/OQL. On the other hand, there are more universal approaches which are unfortunately inconsistent and incoherent.

According to the idea of the stack-based approach (SBA) non-algebraic operators are not indexed by meta-language name/expressions. That is the fundamental difference to those ideas

provided by relational and object-oriented algebras. In SBA there is no need to classify names into "first category" and "second category", as in object algebras, where names of relations are "first category", while names of columns are "second category", i.e. occur only in informal metalanguage indices of operators. In SBA every name has the same semantics and is driven by the scope and binding rules.

Semantics of non-algebraic operators may be expressed using the *eval* procedure introduced previously. Informal description of the corresponding part of the *eval* procedure is given below (θ denotes a non-algebraic operator).

In order to evaluate the query $q_1 \ \theta \ q_2$ the following activities have to be performed:

- Evaluate q_1 and return a collection (bag) of elements.
- For every element *e* that belongs to the returned collection perform the following activities:
  - Calculate the value of the function : *nested(e)*. The result is a set of binders.
  - Insert the calculated set of binders as a new section on the top of the ENVS stack.
  - Evaluate q_2 in this new environment.
  - Calculate the temporary result for *e* by joining it with every result returned by q_2. The function *join* strictly depends on the type of operator θ.
  - Remove the inserted ENVS section.
- Sum all temporary results into the final one. The way how the temporary results will be summed depend strictly on the type of operator θ.

The idea of the function *nested* is to return binders to the interior of an object. For instance, if we have an object $<i_1$, Performer, {$<i_{11}$, *id*, '435'>, $<i_{12}$, *name*, 'johnb'>, ...}>, then *nested*(i_1) = {id(i_{11}), name(i_{12}), ...}. The function is then generalized to arguments being any query results in the following way:

- If r is an identifier of a complex object, then nested returns binders referring to subobjects of this object, as shown above.
- If r is a binder, then nested(r) = {r} (i.e., the result contains this single binder).
- If r is an identifier of a link object, then nested returns a set containing a single binder referring to an object pointed out by r. For example, if the object store contains objects $<i_{28}$, PerformedBy, $i_{10}>$ and $<i_{10}$, Performer, {...} >, then nested(i_{28}) = { Performer (i_{10}) }.
- For structures, nested(**struct**{ v_1, v_2, v_3, ...}) = nested(v_1) ∪ nested(v_2) ∪ nested(v_3) ∪ ...
- For any other cases nested returns an empty set.

The pseudo-code representation of the *eval* procedure is given below:

The detailed description of particular non-algebraic operators is included in the consecutive sub-sections.

```
(1) procedure eval( q : BPQL query)
 (2) begin
 (3)  ...
 (4)  // non-algebraic binary operators
 (5)  case q recognised as  q1 θ q2 (where q1, q2 are queries and θ is a binary non-algebraic operator):
 (6)  begin
 (7)        tmpResBag: bag of Result;    // local variable used to stored the whole set temporary results
 (8)        tmpRes     : Result;         // local variable used to stored a single temporary results
 (9)        finalRes   : Result;         // local variable used to stored the final query results
(10)        singleRes  : Result;         // local variable used to stored a single result of q1
(11)        tmpResBag := ∅;              // for start, the set of temporary results is empty
(12)        eval(q1);                    // call eval procedure for q1, its result will be stored on the QRES top
(13)        for each e in top(QRES) do   // for each single result of q1
(14)          push(ENVS, nested(e))      // creates a new section on the ENVS stack
(15)          eval(q2);                  // call evaluation procedure for q2, its result stored on the top of QRES
(16)          tmpRes := joinθ(e, top(QRES));        // join single result of q1 with the results of q2; it strictly depends on θ
(17)          tmpResBag := tmpResBag union { tmpRes}; // add a single temporary result to the set of temporary results
(18           pop(QRES);                 // remove from QRES the single results of q2 evaluation
(19)          pop(ENVS);                 // remove from ENVS the created section
(20)        end;                         // for each
(21)          finalRes := sumθ(tmpResBag); // sum all temporary results; it strictly depends on θ
(22)          pop(QRES)                  // remove from QRES the results of q1 evaluation
(23)          push(QRES, finalRes)       // store the final result s of the overall query evaluation on the top of the stack
(24)        end;
(25)        .....
(26) end;
```

4.2.7.1 Selection

Selection is used to extract objects of a given type (i.e. selection objects) which satisfy a set of criteria (i.e. selection criteria) . In BPQL selection is expressed as the binary *where* operator and has the following syntax: q_1 *where* q_2. Its left argument is a query to retrieve selection objects. Its right argument is a query represented the selection criteria. The latter query has to return a Boolean value (i.e. either *true* or *false*).

For example, we may use selection if we want to extract a set of activity instances which had been created before the 1st January 2004.

```
ActivityInst where (creationDate < '2004-01-01')
```

According to the *eval* procedure, for every object *e* belonged to the result retrieved by q_1 the *join* function returns:

- one-element bag (i.e. {e}) if the query q_2 operated on *e* returned *true*,

- empty bag (i.e. {∅}) if the query q_2 operated on e returned *false*.

The *sum* function sums (as for sets) all the temporary results.

4.2.7.2 Projection

Projection (or navigation) is used to extract objects which are related to the other objects in someway (e.g. by association or being a part of) . In BPQL projection is expressed by the dot operator and has the following syntax: $q_1.q_2$. The left argument is a query which defines the way how to get to the object s returned by the right argument. According to the *eval* procedure, the *join* function returns the results retrieved by q_2 ignoring the results retrieved by q_1 (i.e. e objects). The *sum* function sums (as for sets) all the temporary results of q_2.

For example we may use projection/navigation to extract all names of performers that participate in any activity instance.

```
((ActivityInst.PerformedBy).Performer).name
```

Such constructs are also known as *path expressions* and can be written without parentheses as:

```
ActivityInst.PerformedBy.Performer. name
```

4.2.7.3 Dependent Join

The dependent join operator is used to retrieve combined values. In BPQL dependent joins are expressed by the **join** operator and has the following syntax: q_1 **join** q_2. The left and right arguments represent values (in particular, references) which are to be associated.

For example, we may use a dependent join to retrieve objects which include information about activity instances performed by *johnb* and all the states in which these activity instances remained/remain.

```
((ActivityInst where (PerfromedBy.Performer.name) = 'johnb') as P).(P join (P.ProcessedIn.ActivityInstState))
```

According to the *eval* procedure, the *join* function merges the object e and every e_2 object retrieved by q_2 into a new structure. A temporary result includes all such structures created for the given object e and all corresponding e_2 objects. The *sum* function sums (as for sets) all the temporary results.

4.2.7.4 Existential Quantifier

Existential quantifier is used to check whether exists at least one value which satisfies the given condition In BPQL existential quantifier is represented by the **for some** operator and has the following syntax: **for some** q_1 (q_2). Its left argument defines a collection of values which are checked. Its right argument represents the query representing the condition. Traditionally, we use this operator in the prefix syntax rather that in the infix syntax supposed for other non-algebraic operators.

For example, we may use the existential quantifier to check whether exist an activity instance started before 1st of January 2004 which has no been finished yet (i.e. its current operational superstate is different than '*Finished*').

```
for some (ActivityInst as ai)
(ai.creationDate < '2004-01-01'  and (ai.currState.opSuperState) <>'Finished')
```

According to the *eval* procedure, the *join* function returns all the values returned by q_2 ignoring values returned by q_1 (i.e. *e* objects). The *sum* function returns:

- *true* if at least one temporary result returned from q_2 is true,
- *false* if all the temporary result returned from q_2 are false.

4.2.7.5 Universal Quantifier

A universal quantifier is used to check whether the given condition is satisfied for all the checked values. In BPQL a universal quantifier is expressed by the **for all** operator and has the following syntax: **for all** q_1 *(*q_2*)*. Its left argument defines a collection of values which are checked. Its right argument represents the condition that has to be satisfied by all the values. As in mathematics, a universal quantifier is dual to the existential one and connected with it by the De Morgan's law:

- **not (for all** q_1 (q_2)) is equivalent to **for some** q_1 (**not** q_2))
- **not (for some** q_1 (q_2)) is equivalent to **for all** q_1 (**not** q_2))

For example, we may use a universal quantifier to check whether all activity instances started before 1st of January 2004 has already been finished (i.e. its current operational superstate is '*Finished*').

```
for all (ActivityInst as ai)
( ai.creationDate < '2004-01-01' and (ai.currState.opSuperState) = 'Finished')
```

According to the *eval* procedure, the *join* function returns all the values returned by q_2 ignoring values returned by q_1 (i.e. *e* objects). The *sum* function returns:

- *true* if all the values returned by q_2 are true,
- *false* otherwise.

Note that according to the De Morgan law, if q1 returns an empty set, then

for all q_1 (q_1)

returns *true* independently of the result returned by q_2 (actually, there will be no such a result). This warning is essential due to very frequent errors with the use of universal quantifiers in such cases. For example, many programmers think that

for all (**bag**(1,2) **intersect bag**(3,4)) **as** x (2*2 = 5)

should return *false*, what is wrong: the result of the above query is *true*.

4.2.8 Sorting Operator

Sorting operator is a non-algebraic operator combined with some additional processing. It sorts a collection returned by a BPQL query and has the following syntax: q_1 **order by** q_2. Both arguments are BPQL queries. The left arguments returns a collection of objects which are to be sorted. The right argument determines the key which will be used to sort the returned objects. The result of the **order by** operator is a sequence.

Processing of an **order by** query is as follows:

- a dependent join: q_1 join q_2 is evaluated. If q_1 returns a bag: **bag**(r_1, r_2, …) and q_2 returns a structure: **struct**{v_1, v_2, ..., v_k} then the whole query returns the following bag:

  bag{ **struct**{ r_1, v_{11}, v_{21}, ..., v_{k1} },
  struct{ r_2, v_{12}, v_{22}, ..., v_{k2} }, **.... }**
- if any of v_{ij} is a reference, its value is automatically extracted (using the *deref* function).
- the bag returned by the whole query is sorted (first key, then second key, etc.). This operation results in creation of a new sequence:

  sequence{ **struct**{ s_1, vs_{12}, vs_{21}, ..., vs_{k1} },
  struct{ s_2, vs_{12}, vs_{22}, ..., vs_{k2} }, ... }

  s_i represents a sorted r_i value.
- the final query results is a sequence of s_i values: **sequence**{ s_1, s_2, ...}.

4.2.9 Imperative Constructs

Imperative constructs enable WfM systems to express more complex requests which require some programming features. Together with procedures and functions, these constructs may be used to simplify BPQL queries. In the consecutive sections we present a set of basic imperative constructs available in BPQL.

4.2.9.1 Statement

A statement is a programming instruction. Every imperative construct is treated as a statement. The simplest statement is an assignment.

4.2.9.2 Statements Block

A statements block is a set of statements which should be considered and executed as a group. This set may also consist of a single statement. If a statement block includes more than one statement two additional syntactic tokens are used to begin ({) and to end such a block (}).

4.2.9.3 Creating New Object

When we need to create a new object or a bag of objects we may use the *create* construct. This operator had the following syntax: **create** *query*. The result of this construct is a reference (or a bag of references) to the new objects that are determined by *query*. The names of newly created objects are described by the binders of this query.

For example we may use the *create* operator to create local copies of objects of the finished process instances and name them using the *FinProcessInst* alias.

```
create (ProcessInst where (currState.opSuperState) = 'Finished') as FinProcesInst;
```

4.2.9.4 Update, Insert and Delete Operators

In the same way as above we can define other simple imperative constructs such as assignment (update), insert and delete. Syntactic and semantic details of these operators can be found in [Subieta04] .

4.2.9.5 For-Each Operator

A for-each operator is used to process a collection of values returned by a query; one after another. In BPQL for-each operator is expressed by the **for each** keyword and has the following syntax: **for each** *query* **do** *statementBlock*. The query returns a collection of values and the statement block is executed for every value returned within the collection.

For example we may use the **for each** operator together with the assignment operator (see the next section) to change the state of all the activity instance which have been delayed more than 20 days. [4]

```
for each (ActivityInst where
        (DiffDate(CurrDate(), deadline, 'd') > 20) and (currState.opSuperState) = 'Running')
 do currState.opState := 'Suspended';
```

The *eval* procedure works in the same way as non-algebraic operators. Semantics of this operator is defined as follows:

- Evaluate q_1, its result is stored on the top of QRES.
- For every element *e* belongs to the set of results returned by q_1:
  - Push *nested(e).* on the top of ENVS;
  - Execute statement block;.
  - Pop ENVS;.
- Pop QRES.

[4] Here and in the rest of the thesis we assume that there is a DiffDate built-in function. This function returns the difference (distance) between two dates (date1 – date2). This difference is returned in the time unit specified as the third argument of the function call. The 'd' time unit means that the difference will be returned as the number of days. This value will be rounded.

4.2.9.6 Control Flow Operators

Control flow operators are used to change the sequence of executed statements. In BPQL there are two control flow operators, namely if-then-else operator and switch-case operator.

If-then-else operator has the following syntax: **if** *query* **then** *block-of-statements*$_1$ **else** *block-of-statements*$_2$. The query has to return a Boolean value. If the returned value is *true*, then *block-of-statements*$_1$ is executed. Otherwise *block-of-statements*$_2$ is executed. There is also a simplified version of this operator: **if** *query* **then** *block-of-statements*$_1$. Its meaning is the same as for appropriate parts of the previous form.

An example of how this operator may be applied is given below. If *johnb* has no task assigned, the query returns the Resource object represented *johnb*. Otherwise it returns the resource with the minimal number of tasks currently assigned:

```
if
        for some (ActivityInst as a) ( a.currState.opSuperState = 'Running' and
                a.PerformedBy.Performer.name  = 'johnb')
then
        return Resource  where (
                count(isPerformer.Performer.Performs.ActivityInst where
                        (currState.opSuperState) = 'Running') =
                min(Resource.count(IsPerformer.Performer.Performs. ActivityInst where
                        (currState.opSuperState) = 'Running'))
                );
else
        return Resource where name = 'johnb';
```

Switch-case operator has the following syntax: *switch* ***(****query****)*** *do list-of-cases*. The query has to return a single value *s* of a numeric, Boolean or string type. The *list-of-cases* includes one of more *cases*. A single case has the following syntax: *case value****:*** *block-of-statements*. If the value is equal to *s*, then the *block-of-statements* is executed. After that the other cases are checked against the value. To leave this operator we have to use the **break** operation. It may be used at any point of the switch-case operator.

For example we may use assignment operator to change the state of the all activity instances which have been delayed more than 60 days. In addition if they are performed by the automatic agent (i.e. system), they are completed. Otherwise they are terminated.

```
for each (ActivityInst where (DiffDate(CurrDate(), deadline, 'd') > 60) and
                    (currState.opSuperState) = 'Running')
          as AI))
 do {
          AI.currState.opSuperState    := 'Finished';
          switch (AI.PerformedBy.Performer.name) = 'system') do {
                    case true: {
                              AI.currState.opState := 'Completed';
                              break;
                    }
                    case false: {
                              AI.currState.opState := 'Terminated';
                              break;
                    }
          }
   }
```

4.2.9.7 Loop Operators

Loop operators are used to perform repeatable actions. In BPQL there are two loop operators, namely *while-do* operator and *do-while* operator.

While-do operator has the following syntax: **while** ***(****query****)*** **do** *block-of-statements*. The query has to return a Boolean value. If it returns *true*, the block of statements is executed. Then the query is re-evaluated and if returns *true*, the block of statements is executed again. This loop is continued until the query returns *false*.

Basically, *do-while* operator is similar to the previous one. It has the following syntax: **do** *block-of-statements* **while** ***(****query****)***. However, in this operator first the block of statements is executed and then the query is checked. If it returns *true* the process is repeated.

In addition to these operators two other operators need to be introduced: **break** and **continue**. The former operator is used to leave immediately a given loop. The latter operator is used to finish a single circle of a given loop.

The above operators may be used for example to remain only two hundreds delayed activity instances with the minimal delay:

```
while (count(ActivityInst where (currState.opSuperState) = 'Running'
        and (deadline < CurrDate())) > 200)
do {
        for each ((ActivityInst where (deadline =
                                min((ActivityInst where (currState.opSuperState) = 'Running
                                and deadline < CurrDate()).deadline)
                as AI)
         do {
                AI.currState.opSuperState   := 'Finished';
                AI.currState.opState        := 'Terminated';
        }
}
```

4.2.10 Procedures and functions

BPQL provides mechanisms to define procedures and functions. This mechanism may be used, in particular, to simplify complex queries and to encapsulate repeatable query parts. A *procedure* is a block of statements which are executed in the context of parameters which may be processed. A procedure may have parameters. Every parameter may be an input parameter or an output parameter (with *out* keyword). The former is a call-by-value parameter that can only be read in the procedure body. The latter is a call-by-reference parameter and may be used to modify objects that are kept outside the procedure local environment. The procedure body consists of one or more statements. Every statement ends with semicolon. The *return* operator may be treated as a special statement which finishes the procedure immediately.

A *function* is a procedure which may return a value (perhaps, complex and being a collection) compatible with results of queries. Semantics for a procedure/function call is defined in the *eval* procedure as follows:

- Bind the procedure name on ENVS. As the result we get the reference to the procedure.
- Initialise the procedure call. ENVS is extended with a new section (an activation record) having the following information:
  - binders storing procedure parameters,
  - binders to local objects of the procedure,
  - a return trace to come back to the caller code after terminating the procedure.
- Evaluate procedure parameters, which are also BPQL queries. The evaluation results are stored on the top of QRES.

- Store procedure parameters binders in the activation record according to their values stored at QRES. Remove these values from QRES.
- If local objects were declared, they are stored as volatile objects. Their binders are inserted into the activation record on ENVS.
- The procedure body is executed. Queries in the procedure body have access to all ENVS binders, including volatile objects and parameters of the procedure.
- If the procedure control reaches a return statement, then it finishes. If there was a return value defined as a BPQL query, it is evaluated and its result is stored on the top of QRES.
- The procedure is finished. All volatile objects are being removed. ENVS is cleared from the activation record. The system returns to the point of program according to the return trace. The procedure result is stored at the top of QRES.

In the stack based approach implemented in BPQL all procedures may be recursive.

An example of a functional procedure which returns a set of start activities for a given process definition is given below:

```
procedure StartActivities (parProcessDef) {
  return parProcessDef.ConsistsOf.Activity where count(From) = 0;
}
```

4.2.10.1 Predefined Context-dependent Functions

In addition to the constructs described in the previous sections, we also need in BPQL some functions which would extract the context of evaluated queries. That feature is especially important when we would like to define a query which refers to the workflow objects which are currently processed such as process instances or activity instances. For example, if we want to define in a workflow process a workflow participant assignment in which a given activity instance, say A, will be performed by the same person who performed the previous activity, we need somehow refer to the current activity instance. Since A is created at the run-time, at the definition stage we are not able to refer to a particular A precisely. What we may do is to use a function which will return A at the run time.

Context dependent function operate on Workflow Expression Evaluation object which so far includes two context attributes, namely *processInstId* and *activityInstId* attributes. Thus there are two context functions:

- *ThisProcessInst* which returns the reference to the process instance which identifier is equal to the *processInstId* attribute.
- *ThisActivityInst* which returns the reference to the activity instance which identifier is equals to the *activityInstId* attribute.

There is no obligation on the WfM system to set the mentioned context attributes before execution on every BPQL query. If the context attributes are not set, a context dependent function throws an exception.

4.3 Pragmatics – by Example

Pragmatics describes how a given language should be used. Pragmatics is usually expressed by set of patterns, rules or cases which should be either used (positive cases) or avoided (negative cases).

Following this approach for every introduced operator we included an example how it may be used. In addition to that we present in this section a comprehensive example how various elements of BPQL may be used together to satisfy practical needs in a workflow process definition. To make it readable, first we specify the needs and then show how they may be formulated by a BPQL query. Description of the BPQL query is pretty detailed and concerns its every significant part.

4.3.1 Process for Ordering Laptops

Let us assume that there is a simplified version of a process for ordering laptops. Every registered customer may order any number of laptops. An order made by a customer is then accepted by a company seller which is responsible for verifying financial status of the customer and ability to meet the order at the requested time. For a bigger order, the order is served by a senior-seller. Otherwise, it is served by a plain seller which has the minimal work-load (i.e. a person with minimal number of tasks assigned). If the order was accepted, it is sent to the production department for completing. It is an assumption that all the orders are processed by the company.

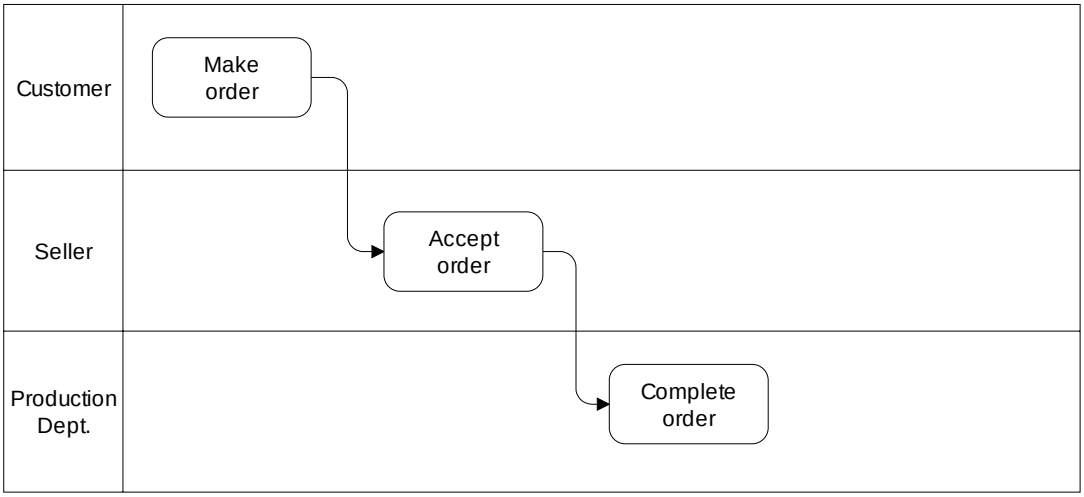

Figure 4.8 A simplified model for ordering laptops[5]

Even in this simplified example some of the requirements for the process have to be defined on the basic of process execution data. For instance, the 'minimal work-load' requirement may be only expressed using a query on the current task assignment. Yet, to select a kind of a seller for accepting the order we have to define as a condition on process relevant data (workflow environment). The requirement may be easily expressed using BPQL. What is more, their definition in BPQL simplifies the definition of the process making it more generalised. Instead of defining two activities for accepting the order: one for 'Senior-seller' and another for 'Seller', it is possible to define only one for both mentioned workflow participants. Such an approach seems to be more adequate for real computerised business processes. The role 'seller' may be defined in BPQL as follows:

```
(1) ( if
(2)       for some (ThisProcessInst.Has.ContainerAttribute
(3)                 where name = 'Order.Value' as A) ((int)(A.value) > 30000)
(4) then
(5)       'Senior-Seller' as r
(6) else
(7)       'Seller' as r
(8) ).
(9)  Resource where (position = r and
(10)    count(IsPerformer.Performer.Performs.ActivityInst
(11)          where (currState.opSuperState) = 'Running')
(12) =
(13)   min((Resource  where position = r).
(14)    count(IsPerformer.Performer.Performs.ActivityInst where
(15)      (currState.opSuperState) = 'Running'))
```

[5] The process has been modelled using Business Process Modelling Notation described in [BPMN04].

The first part of the query (lines 1-8) is to select the position of the requested performer. In the next part (line 9) this value is used to select the users (resources) who are employed at the selected position. Then (lines 10-11) for every selected resource the number of tasks which are currently performed (i.e. remain in the *Running* superstate) by this resource is evaluated. This value is finally compared to the minimal number of the assigned tasks for the all calculated resources employed at the position (lines 13-15). Such construct (lines 10-15) gives the opportunity to choose a resource employed at the position and having the minimal current work load.

5 Built-in Workflow Monitoring Functions

The BPQL operators introduced in the previous chapter are able to express most, if not all, useful workflow queries which operate on the metamodel. However, from the practically point of view these queries may be quite complex and include many repeatable fragments. Definition of the same fragments many times may be quite annoying and cause additional mistakes.

For example, if we want to extract information about the second-successors of the currently processed activity instance we have to build a query which consists of two parts. The first part extracts the direct successors of a given activity instance (i.e. the current activity) while the second one extracts the direct successors of a set of activity instances which are returned by the first part.

```
(1)  ThisActivityInst.                    // extract the current activity (use function described in 4.2.10.1)
(2)  To.TransitionInst.To.ActivityInst.   // extract its direct successors
(3)  To.TransitionInst.To.ActivityInst    // extract the direct successors of its direct successors
```

As may be noticed the lines 2 and 3 are identical and express the procedure of retrieving the direct successors for a bag of activities (it may also consist of one element). Instead of repeating line 2 and 3, it is possible to define a procedure which as the argument takes a bag of activities and retrieve their direct successors as the result.

```
(1) procedure DirectSuccActivityInst(parActivityInst) {
(2)   return parActivityInst.To.TransitionInst.To.ActivityInst;
(3) }
```

In this case the original query may be simplified and has the following form:

```
DirectSuccActivityInst(DirectSuccActivityInst(ThisActivityInst))
```

In order to avoid writing such repeatable fragments we propose to define a set of built-in workflow monitoring functions. As it is presented in Figure 5.1 these functions are defined on the top of the existing BPQL operators and provide more ‘advanced’ workflow-specific operators to build BPQL queries.

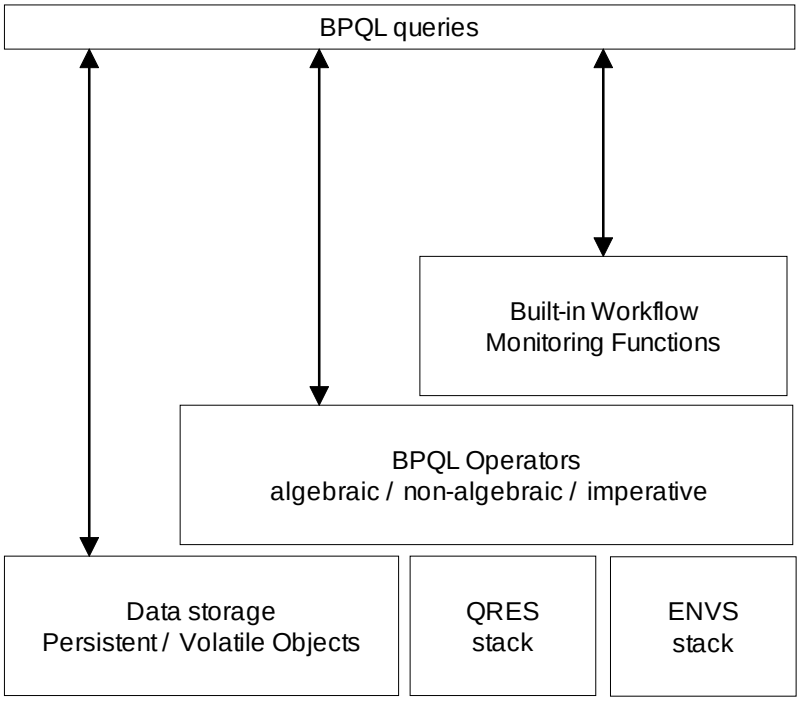

Figure 5.1 BPQL layered architecture

The BPQL built-in workflow monitoring functions have many advantages, the most important are:

- query simplification – as was stated earlier BPQL functions give a chance to reduce the repeatable fragments of queries and, as a consequence, make them easier to define and analyse.
- quality improvement – using BPQL functions we also reduce the chance of introducing inefficiency or errors in our queries. Since BPQL functions are used in various queries, usually they are well tested and optimised.
- effort reduction – especially for complex queries BPQL functions may shorten development time and enable their designers to focus on the actual goal, integrating already existing solutions.
- BPQL enrichment – by defining new BPQL functions we may enrich the language providing the more advanced functionality. Such functionality may be offered to the other designers/companies as BPQL extensions.

Since BPQL functions operate on the workflow metamodel (especially on its execution part) they are regarded as a set of monitoring functions. These functions have been categorized according to functional parts of a WfM system which they cover. So far three types of BPQL functions have been identified: process flow functions, workflow participant assignment functions, and quality of service functions.

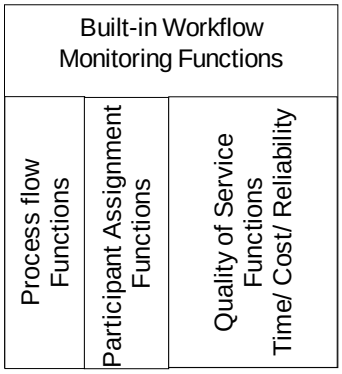

Figure 5.2 Types of the built-in workflow monitoring functions

All the mentioned types of monitoring functions are described in the consecutive sections. For every type we present its main objectives and describe the individual functions. If required, we make further categorisation of the functions within a given type. For every function we also describe its main objectives, specify its parameters and results as well as identify the entities of the metamodel which this function operates on. For the functions that introduce a new way of processing the workflow entities or require more advanced BPQL queries we also specify their algorithms and describe their full BPQL code. All together we describe 32 built-in workflow monitoring functions. They are listed in Table 5.1.

The BPQL code of the all functions described in this chapter may be found in Appendix B. Moreover, at the end of the chapter we also present some practical examples how these functions may help in definition of dynamic workflow participant assignment, selection of the best performers, identification of the process failures as well as verification of the reachability property.

Table 5.1 Specification of the built-in workflow monitoring functions

Function name	Short description
Process Flow Functions	
StartActivity	start activities for given process definitions
EndActivity	end activities for given process definitions
SuccActivity	the successors of given activities
PredActivity	the predecessors of given activities
StartActivityInst	the start activity instances for given process instances
EndActivityInst	the end activity instances for given process instances
SuccActivityInst	the successors for given activity instances
PredActivityInst	the predecessors for given activity instances
Workflow Participant Assignment Functions	
WorkloadCount	the current amount of tasks (activity instances) assigned to given resources

WorkloadExpectedTime	the total time needed to complete tasks assigned to given resources
WorkloadExpectedCost	the overall expected cost to complete the tasks assigned to given resources
WorkloadAccPriority	the accumulated priority of the tasks assigned to given resources
BestPerformerCount	the resources with the minimal current workload in terms of the number of the assigned tasks
BestPerformerExpectedTime	the resources with minimal current workload in terms of expected total time to complete the assigned tasks
BestPerformerExpectedCost	the resources with minimal current workload in terms of the expected overall cost to complete the assigned tasks
BestPerformerAccPriority	the resources with minimal current workload in terms of the accumulated priority to complete the assigned tasks
Quality of Service functions	
ActivityInstStateDuration	a sum of periods which activity instances spent in given operational states/superstates
ActivityInstProcessingTime	the expected processing time needed to complete given tasks
ActivityInstWaitingTime	the waiting time for given tasks
ActivityInstWorkingTime	the working time for given tasks
ProcessInstWaitingTime	the waiting time for given process instances
ProcessInstWorkingTime	the working time for given process instances
ActivityWaitingTime	the average value of the total waiting time for given activities
ActivityWorkingTime	the average value of the total working time for given activities
ProcessWaitingTime	the average value of the total waiting time for given process definitions
ProcessWorkingTime	the average value of the total working time for given process definitions
ActivityInstCost	the cost for given activity instances
ProcessInstCost	the cost for given process instances
ActivityCost	the average cost for given activities
ProcessCost	the average cost for given process definitions
ActivityReliability	the reliability for given activities
ProcessReliability	the reliability for given process definitions

5.1 Process Flow Functions

The main aim of process flow functions is to simplify operations on process graphs. These operations includes finding the start and end activities, and determining predecessors or successors, direct or all, for one or more activities. The process flow functions are provided for process definition and process instantiation.

5.1.1 Process Definition Functions

These functions operate on *Activity* and *Transition* objects as well as *From* and *To* links between these objects.

5.1.1.1 Start Activities

This function is to retrieve a bag of activities which are regarded as the start activities of given process definitions.

A *start activity* is defined as an activity which does not have any ingoing transition. In terms of the metamodel a start activity is the *Activity* object which belongs to a given *ProcessDef* object (usage of *ConsistsOf* link and *BelongsTo* pointer) and for which the number of ingoing *Transition* objects is zero (i.e. count(Activity.From) = 0).

This function written in BPQL looks as follows:

```
(1) procedure StartActivity(parProcessDef) {
(2)  return (parProcessDef as e).
(3)       (
(4)                 (e as resProcessDef),
(5)                 ((e.ConsistsOf.Activity where (count(From)=0)) as resStartActivity)
(6)       );
(7) }
```

As the function result we return the bag of argument process definitions (line 4) and for every such definition a bag of its start activities (line 5). The elements of the former bag are available through the *resProcessDef* alias. The elements of the latter bag are available through *resStartActivity* alias. In order to achieve such structure of the result we initially aliased the argument bag of process definitions and then used projection for processing of every its element (line 2).

5.1.1.2 End Activities

This function is to retrieve a bag of activities which are regarded as the end activities of given process definitions.

An *end activity* is defined as an activity which does not have any outgoing transition. In terms of the metamodel an end activity is the *Activity* object which belongs to a given *ProcessDef*

object and for which the number of outgoing *Transition* objects is zero (i.e. count(Activity.To) = 0). The BPQL definition of the end activity function uses the similar constructs as those applied in the function to retrieve start activities.

5.1.1.3 Successor Activities

This recursive function is to retrieve a bag of activities which are successors of given activities[6]. It works in two modes. In the first mode ('all') it returns all successors. In the second mode ('direct') it returns only direct successors.

Activity A2 is defined as a *successor* of activity A1 if:

- A2 is a direct successor of A1, or
- there is an activity A3 which is a direct successor of A1, and A2 is a successor of A3.

Activity A2 is a *direct successor* of activity A1 if there is a transition from A1 to A2. In terms of the metamodel, for a direct successor we have the following dependency:

A1.To.Transition.To.A2

The recursive algorithm to find all successors for a given activity is defined as follows:

1)	There is a bag S of activities for which we would like to find successors
2)	If the bag is empty, then return (for the whole function or for a given call – see step 4)
3)	Find the direct successors of the activities included in S.
4)	Find successors of the activities found in step 3. This is expressed by a new call of the function (Step 1).
5)	Return the activities found in steps 3 and 4 (for the whole function or for a given call – see step 4).

On the basis of the above algorithm we are able to define in BPQL the successor function. It is presented below. Since the function is recursive first we check the stop condition (line 2) and if it is satisfied (i.e. no activities to process) the function returns an empty bag (line 3). Otherwise, we check what is the kind of the requested successors (line 4). If the kind is 'all', the function determines both the direct successors of the activities (*parActivity* argument) (line 5) and the remaining successors by using a recursive call to the *SuccActivity* function with the bag of the determined direct successors as the function arguments (line 6). If the kind is 'direct', the function returns only direct successors (line 8). By default, the function returns an empty bag (line 9).

[6] This function does not cope with loops occurring in process graphs. To deal with this problem we have to keep the list of successors already found and check if newly reached activities are not already on the list.

```
(1) procedure SuccActivity(parActivities, parKind) {
(2)  if (count(parActivities) = 0) then
(3)       return bag();
(4)  switch (parKind) {
(5)       case 'all'      : {  return distinct((parActivities.To.Transition.To.Activity as p).
(6)                                        (p union SuccActivity( p, parKind)));
(7)       }
(8)       case 'direct'  : {  return distinct(parActivities.To.Transition.To.Activity);  }
(9)       case default: {  return bag(); }
(10)   }
(11) }
```

5.1.1.4 Predecessor Activities

This recursive function is to retrieve a bag of activities which are predecessors of given activities. This function also works in two modes: all and direct predecessors.

Activity A2 is defined as a *predecessor* of activity A1 if:

- A2 is a direct predecessor of A1, or
- there is an activity A3 which is a direct predecessor of A1, and A2 is a predecessor of A3.

Activity A2 is a *direct predecessor* of activity A1 if there is a transition from A2 to A1. In terms of the metamodel, for a direct predecessor we have the following dependency:

```
A2.From.Transition.From.A1
```

The BPQL definition of the predecessor function uses the similar concepts as those used in the successor function.

5.1.2 Process Enactment Functions

These functions are similar to the process definition functions but operate the process enactment part of the metamodel, especially on *ActivityInst* and *TransitionInst* objects as well as *From* and *To* links between these objects.

5.1.2.1 Start Activity Instance

This function is to retrieve the start activity instances for given process instances. There is only one start activity instance for a given process instance. A *start activity instance* is defined as an instantiation of a start activity (*Activity* object, *InstantiateAs* link) within a given process instance (*ProcessInst* object, *BelongsTo* link).

In BPQL the start activity instance function is defined as follows:

```
(1) procedure StartActivityInst( parProcessInst) {
(2)   return (parProcessInst as e).
(3)                (
(4)                   (e as resProcessInst),
(5)                   ((e.ConsistsOf.ActivityInst
(6)                            where (InstanceOf.Activity.(count(To) = 0))
(7)                    ) as resStartActivityInst)
(8)                );
(9) }
```

Firstly, for every process instance passed as the function argument, we determine all its activity instances (line 5). Then, we return only these activity instances which are instantiations of the start activity (ies) (line 6).

5.1.2.2 End Activity Instances

This function is to retrieve a bag of the end activity instances for given process instances. For a given process instance there may be no end activity instance yet (i.e. none of the end activity has been instantiated yet), one or more end activity instances. An *end activity instance* is defined as an instantiation of an end activity (*Activity* object, *InstantiatedAs* link) within a given process instance (*ProcessInst* object, *BelongsTo* link).

The BPQL definition of the end activity instances function uses the similar concepts as those used in the start activity instance function.

5.1.2.3 Successor Activity Instances

This recursive function is to retrieve a bag of activity instances which are successors of given activity instances. This function works in two modes: all and direct successors.

The definition of the activity instance successor and direct successor is similar as that described for activity. The only difference is that *ActivityInst* and *TransitionInst* objects are used instead of *Activity* and *Transition* objects respectively.

5.1.2.4 Predecessor Activity Instances

This recursive function is to retrieve a bag of activity instances which are predecessors of given activity instances. This function also works in two modes: all and direct successors.

The definition of the activity instance predecessor and direct predecessor is similar as that described for activity. The only difference is the same as for the activity instance successor function.

5.2 Workflow Participant Assignment Functions

These functions are used for selection of the 'best' workflow participants which may perform given activities. There are various criteria which define what 'best' means. The most popular techniques are based on calculation of the number of the assigned tasks, their overall cost, or the total time required to perform them. The other, more advanced ones, may be connected with risk calculation or time constraints prediction.

In this section we provide a sample set of basic functions to select workflow participants. This set consists of two groups of functions: workload functions and best performer selection functions. The former functions determine some business factors for all workflow participants. Based on these factors the latter functions are able to select the 'best' participants, those with the most optimal values for the calculated factors. The presented set may be extended of more advanced techniques.

5.2.1 Workload Functions

As was stated earlier these functions are to provide basic business factors for individual resources or workflow participants. Since these factors use information about the current participant workload, they hold only for a given point of time. The list of the provided factors for the assigned activities includes: the current amount, total processing time, the overall cost required to complete them and their accumulated priority.

The functions operate on *ActivityInst*, *Activity*, *ActivityInstState*, *Performer* and *Resource* objects as well as links amongst them.

5.2.1.1 Current Amount

This function is to calculate the current amount of the assigned activity instances or tasks for given workflow participants.

An assigned task is defined as an activity instance for which the current superstate is 'Created' or 'Running'. The current superstate may be checked using the following query:

```
ActivityInst.currState.opSuperState in ('Created', 'Running')
```

The BPQL function is defined below. For every resource passed as the function argument we calculate the number of its tasks (lines 5 and 6) which are assigned or currently performed (line 6).

```
(1) procedure WorkloadCount( parResource) {
(2)  return (parResource as e).
(3)        (
(4)          (e  as resResource),
(5)           (count(e.IsPerformer.Performer.Performs.ActivityInst
(6)                    where (currState.opSuperState) in('Created','Running'))
(7)           as resCount)
(8)        );
(9) }
```

5.2.1.2 Expected Processing Time

This function is to calculate the expected processing time required to complete all the assigned activities for given workflow participants.

The expected processing time required to complete a given activity instance is defined as the difference between the expected duration of the activity and the current processing time already spent on execution of the activity instance. In BPQL this algorithm may be expressed as follows:

```
ActivityInst.(InstanceOf.Activity.duration – duration)[7]
```

In BPQL the function is defined below. The expected time is calculated according to the above algorithm (line 7) for tasks assigned or currently processed by a given resource (line 5- 6).

```
(1) procedure WorkloadExpectedTime( parResource) {
(2)   return (parResource as e).
(3)        (
(4)          ( e as resResource),
(5)           (sum((e.IsPerformer.Performer.Performs.ActivityInst
(6)                    where (currState.opSuperState) in ('Created','Running')).
(7)                              (InstanceOf.Activity.duration - duration)
(8)            ) as resExpectedTime)
(9)        );
(10) }
```

5.2.1.3 Overall Expected Cost

Similarly to previous function, this function calculates the overall cost required to complete all of the assigned activities for given workflow participants. To calculate the overall cost we use the activity *cost* attribute. The way of calculating the expected cost follows the same already presented pattern for calculation of the expected processing time.

[7] This and the next two functions do not work well for the case when the actual value of a given activity instance attribute (e.g. duration) is greater that the expected value defined in the activity.

5.2.1.4 Accumulated Priority

This function is to determine the accumulated priority of the assigned tasks for given workflow participants. The way of calculation this priority is simpler than the pattern already presented. The priority is defined as a sum of the actual priorities of the assigned activities.

5.2.2 Best Participant Selection

On the basis of the defined workload functions the best participant selection functions are able to choose the workflow participants (one or more) which possess the optimal value (i.e. usually the minimal value) for the calculated factors. All the previously defined factors are considered.

5.2.2.1 Smallest Current Amount

This function is to select workflow participants from the list of given resources who have the minimal current amount of the assigned activities. If there is more than one workflow participant with the same minimal amount, all such workflow participants are returned.

The algorithm to select the mentioned workflow participant is defined as follows:

1) For every resource given as the argument calculate the current amount of activities assigned to it using WorkloadCount function.
2) For every resource compare the current amount of their assigned tasks with the minimal amount of activities determined for all resources.
3) If the current amount and the minimal amount are equal then this resource is a workflow participant with minimal amount of tasks assigned.

The BPQL definition of the function is given below. To perform the 2nd step of the algorithm we had to use *group as* operator to:

- create a single binder for the bag returned by step 1 and use projection to process every element of the bag (line 2)
- determine the minimal amount of the assigned activities by applying *min* function on the binder, i.e. the whole bag (line 4)
- compare this minimal amount with the amount calculated for every single result (*resCount* attribute) included in the returned bag (line 3).

```
(1) procedure BestPerformerCount(parResource) {
(2)   return  (WorkloadCount(parResource) group as rc).(rc where
(3)          resCount  =
(4)          min( rc. resCount)
(5)         ).resResource;
(6) }
```

5.2.2.2 Shortest Expected Processing Time

This function is to selects workflow participants who require the shortest expected processing time to complete all the assigned activities. The function uses the *WorkloadExpectedTime* function. The way of selecting the best participants is similar to the pattern presented in the previous section.

5.2.2.3 Minimal Overall Cost

This function is to selects workflow participants who require the minimal expected overall cost to complete all the assigned activities. The function uses the *WorkloadAccPriority* function and, as the previous functions, uses the same pattern to select the best participants.

5.2.2.4 Smallest Accumulated Priority

This function also repeats the already applied pattern and selects workflow participants for whom the accumulated priority of the all assigned activities is the smallest one. The function uses the *WorkloadAccPriority* function.

5.2.3 Closing Remarks to Workflow Participant Assignment Functions

As shown in this section on representative examples, there are practically infinitely many criteria to select the best assignment of workflow participants to particular activity instances. A concrete criterion depends on the kind of business that the workflow management system supports as well as on the concrete situation in a particular business process. The traditional solutions, such as those described in [Momotko02b], assume some predefined methods based on selected information only (in particular, on a participant role). As was stated earlier, such an approach is not flexible enough, because it is unable to determine complex criteria based on all the information gathered in the workflow metamodel repository. BPQL introduces a totally new quality to the issue. Instead of determining predefined criteria, it offers for the workflow designers a very simple and flexible tool to be used ad hoc in order to define any criterion that fits the best the current business situation.

5.3 Introduction to Quality of Service Functions

QoS management is considered as one of the main techniques to characterise and measure business processes in both qualitative and quantitative manners. This technique enables organisations to define, enact and monitor business processes according to their requirements expressed in business terms.

QoS management operates on QoS factors. Usually these factors are set according to the initial business process requirements during the business process definition. Then, the required QoS factors are compared with those retrieved from the business process execution. Finally, the results

of the comparison are used to improve business processes and make them possible to meet the initial QoS requirements.

Following the concept presented in [Cardoso02] we selected three basic QoS factors for business processes, namely: *time*, *cost*, and *reliability*. These factors are defined on four levels, namely activity instance, process instance, activity and process levels. To calculate the mentioned QoS factors we propose a set of built-in BPQL workflow functions.

5.4 Time Functions

The main aim of these functions is to provide information about the period spent by workflow objects in selected states. There are four types of workflow objects for which the time QoS factors are defined: process and activity (definition phase) as well as process instance and activity instance (enactment phase). These factors are calculated for process and activity as the average values of those evaluated for all their instantiations (i.e. process instances or activity instances respectively). In addition, the QoS time factors for a process instance are calculated as the sum of the QoS factor values evaluated for all its activity instances.

To simplify comparison of QoS factors for different types of workflow objects, we define three basic QoS time factors: expected processing time, working time and waiting time. Their basic meaning has been defined at the activity instance level. *Expected processing time* is defined as a period needed to perform an activity instance. *Waiting time* represents a period when an activity instance waits in the participant work-list in order to be processed. *Working time* represents a period when an activity is being performed by the workflow participant.

5.4.1 Activity Instance Level

At the activity instance level, BPQL provides four functions to calculate: duration of a given state/superstate, expected processing time, waiting time and working time.

5.4.1.1 Duration of a State and Superstate

This function is to calculate the sum of periods which activity instance spent in given states. If a state is the current state, the value: *currentDate()*[8]*-startDate* contributes to the result.

The BPQL definition of the function is presented below. Firstly, for every activity instance passed as the function parameter we extract information of all its states (the past ones and the current one) (line 6). Then we select only those states which satisfy the criteria on state or superstate names passed as the function arguments (lines 8 and 9). Because activity instances can be in some superstates (general) and states (more specific), we assume these two parameters. If any parameter is empty it means that it should not be taken into account (what is equivalent to the bag

[8] We assume that CurrentDate is a built-in function which returns the current timestamp.

with all states/superstates). For every selected activity instance state we calculate its duration (lines 11-13). We assume that if *finishDate* is 0, then the activity is currently performed.

```
(1) procedure ActivityInstStateDuration( parActivityInst, parOpSuperState, parOpState) {
(2)  return (parActivityInst as e).
(3)       (
(4)         ( e as resActivityInst),
(5)         (sum(
(6)                  (e.ProcessedIn.ActivityInstState
(7)                   where
(8)                           (count(parOpSuperState) = 0 or opSuperState in parOpSuperState) and
(9)                           (count(parOpState) = 0 or opState in parOpState)
(10)                 ).
(11)                 ( if (finishDate > 0) then (finishDate – startDate)
(12)                           else (CurrDate() – startDate)
(13)                 )
(14)        ) as resDuration)
(15)      );
(16) }
```

5.4.1.2 Expected Processing Time

This function is to calculate the expected processing time required to complete given activity instances. The calculation algorithm is the same as that one described in section 5.2.1.2 and uses two values: the expected duration of the activity and the current processing time already spent on execution of the activity instance.

5.4.1.3 Waiting time

This function is to calculate the sum of periods which given activity instances spent in the 'Created' superstate. This function is defined as a particular use of the *ActivityInstStateDuration* function.

5.4.1.4 Working time

This function is to calculate the sum of periods of time which given activity instances spent in the 'Running' superstate. As the previous function, it is also defined as a particular use of the *ActivityInstStateDuration* function.

5.4.2 Process Instance Level

At the process instance level, BPQL provides two functions to calculate QoS time factors: waiting and working times. These time factors are calculated as the average value of appropriate times evaluated for all activity instances belonged to a given process instance.

5.4.2.1 Waiting Time

This function is to calculate the aggregated waiting time for given process instances. At the process instance level waiting time is defined as the sum of waiting times determined for all its activity instances (line 5). In BPQL, definition of the function looks as follows:

```
(1) procedure ProcessInstWaitingTime( parProcessInst) {
(2)   return (parProcessInst as e).
(3)         (
(4)           (e as resProcessInst),
(5)           (sum(ActivityInstWaitingTime(e.ConsistsOf.ActivityInst).resWaitingTime) as resWaitingTime)
(6)         );
(7) }
```

5.4.2.2 Working Time

This function is to calculate the aggregated working time for given process instances and uses the same concept as for waiting time.

5.4.3 Activity Level

At the activity level, BPQL also provides two functions to calculate waiting and working time. These factors are defined as the average values of appropriate times evaluated for all finished activity instances.

5.4.3.1 Waiting Time

This function calculates the average value of the total waiting time for the all finished instantiations of given activities. In BPQL, definition of the function looks as follows:

```
(1) procedure ActivityWaitingTime( parActivity) {
(2)    return (parActivity as e).
(3)         (
(4)           (e as resActivity),
(5)           (avg(ActivityInstWaitingTime(
(6)                    e.InstantiatedAs.ActivityInst   where ((currState.opSuperState) = 'Finished')
(7)             ).resWaitingTime) as resWaitingTime)
(8)         );
(9) }
```

For every activity passed as the function argument we:

- calculate waiting time for all its finished instantiations (line 6),
- determine the average value of all the calculated waiting times (lines 5-7).

5.4.3.2 Working Time

This function is to calculate the average value of the total working time for given activities and is based on the same concept as was used for evaluation of the average waiting time.

5.4.4 Process Level

At the process level, BPQL also provides two functions to calculate waiting and working time. These factors are calculated as the average value of appropriate times evaluated for all process instances of given process definitions.

5.4.4.1 Waiting Time

This function is to calculate the average value of the total waiting time for all finished instantiations of given process definitions. To evaluate this value we use the *avg* function as well as the waiting time function defined at the process instance level. The concept of this function is very similar to that one introduced for calculating of the waiting time at the activity level.

5.4.4.2 Working Time

Also for this function which is to calculate the average value of the total working time for all finished instantiations of given process definitions we use the similar concept as that introduced for calculating working time at the activity level.

5.5 Cost Functions

The main aim of these functions is to provide information about the cost of workflow objects. The basic cost is calculated for activity instances. This cost is calculated on the basis of information about the activity performer-hour rate and time he/she spent on processing a given activity instance (i.e. working time). Then this basic cost is used to calculate the cost of process instance as the total cost of all its activity instances. The cost for activities and processes is calculated on the basis of information about the cost of all the population of their instances. This is the average cost.

5.5.1 Activity Instance Cost

This function is to calculate the cost of given activity instances. The BPQL definition of the function is given below. For every activity instance passed as the function argument we:

- evaluate its working time using *ActivityInstWorkingTime* function (line 5)
- determine the hour rate for the performer of the activity instance (line 6)

- calculate the cost as the product of activity instance performer-hour rate and time spent on its processing (lines 6, and 7). Since working time is expressed in milliseconds before calculation it needs to be converted into hour units (line 7).

```
(1) procedure ActivityInstCost( parActivityInst) {
 (2)   return (parActivityInst as e).
 (3)        (
 (4)           (e as resActivityInst),
 (5)           (ActivityInstWorkingTime(e).
 (6)                    (resActivityInst.PerformedBy.Performer.Represents.Resource.hourRate *
 (7)                     resWorkingTime / (1000*60*60))
 (8)           as resCost)
 (9)        );
(10) }
```

5.5.2 Process Instance Cost

This function is to calculate the cost of given process instances. The BPQL definition of the function is given below. The cost for a single process instance is calculated as the total cost of all its activity instances using the *ActivityInstCost* function (line 5).

```
(1) procedure ProcessInstCost( parProcessInst) {
(2)   return (parProcessInst as e).
(3)        (
(4)           (e as resProcessInst),
(5)           (sum(ActivityInstCost(e.ConsistsOf.ActivityInst).resCost) as resCost)
(6)        );
(7) }
```

5.5.3 Activity Cost

This function is to calculate the cost for given activities. The cost of an activity is calculated as the average cost of all its instantiations (to compare for process instance we calculate the total cost not the average one). This function also uses the cost function defined at the activity instance level and applies the similar pattern as that presented in the previous example.

5.5.4 Process Cost

This function is to calculate the average cost for given process definitions. It uses the same concept as was applied in the previous function.

5.6 Reliability Functions

Reliability is defined at two levels: activity and process levels. Reliability corresponds to the likelihood that activity or process will be performed correctly. Reliability is a function of the failure rate. It is defined as *R(a) = 1 – failure rate*.

The failure rate is defined as the ratio between the number of activity instances (or process instances) which failed (i.e. had been terminated and achieved the *'Finished.Terminated'* state) and the number of activity instances (or process instances) which were performed correctly (i.e. achieved the '*Finished.Completed'* state).

5.6.1 Activity Reliability

This function is to calculate the reliability for given activities. The reliability is returned as a float value between 0 and 1. The BPQL definition of the function is given below. For every activity passed as the function argument first we determine the number of activity instantiations which have been terminated (line 7) and then those which were completed correctly (line 8). Finally, we calculate the activity reliability (lines 6, 7, and 8).

```
(1) procedure ActivityReliability( parActivity) {
(2)  return (parActivity as e).
(3)       (
(4)          (e as resActivity),
(5)          (e.(
(6)                  1 –
(7)                  count(InstantiatedAs.ActivityInst where ((currState.opState) = 'Terminated'))/
(8)                  count(InstantiatedAs.ActivityInst where ((currState.opState) = 'Completed'))
(9)           ) as resReliability)
(10)      );
(11) }
```

5.6.2 Process Reliability

This function uses the same concept as introduced in the previous function and is to calculate the reliability of given process definitions.

5.7 Application Examples

The built-in workflow monitoring functions described in this chapter simplify or solve many real problems related to business process management. For instance, these functions may be applied to monitor QoS parameters, express dynamic requirements on process definition, and detect the most unreliable activities. In this section we present some real examples of their applications. For every

application first we specify the requirement, and then we show how it may be satisfied using the mentioned built-in functions. Majority of these examples concern selected requirements on the process for ordering laptops. Its simplified version has already been presented in the previous chapter. Now, its more detailed version is described in the consecutive sub-section.

5.7.1 Ordering Laptops – Basic Example

Every registered customer can make an order for any number of laptops. Such order is then accepted by a company seller which is responsible for verifying financial status of the customer and ability to meet the order at the requested time. The seller is selected from the list of available sellers as a person which has the minimal work-load (i.e. in this case a seller with minimal number of tasks currently assigned). In addition, if the value of the order is greater than 30k EUR then the order is also accepted by a senior seller.

If the order was not accepted (e.g. financial status of the customer prevent ordering laptops), appropriate information is sent to the customer and the ordering process is terminated. Otherwise, the order is sent to the production department for completing and to the customer informing him/her about the order acceptance and the possible delivery date. In parallel to the laptop production, the order owner is completing the required shipment documents. The order owner is defined as a person who made the final acceptance of the order.

When the laptops are ready and the shipment documents are prepared, the shipment department sends the laptops to the customer.

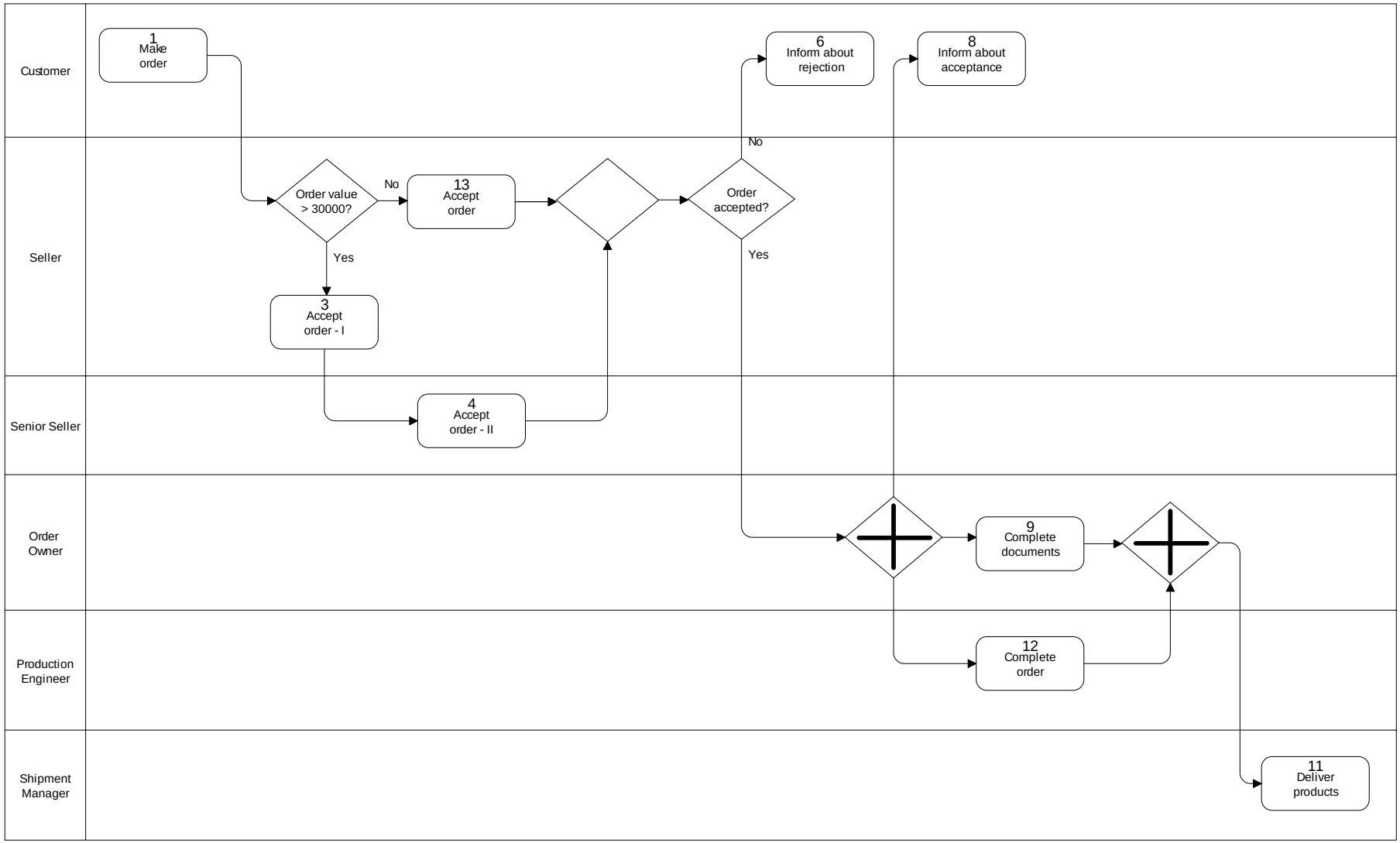

Figure 5.3 The process model for ordering laptops

5.7.2 Workflow Participant Assignment

There are several requirements for right workflow participant assignment included in the description of the process for ordering laptops.

The first requirement is related to selection of the customer to inform him/her about either rejection or acceptance of the order. In terms of workflow processes it means that the workflow participant assigned to the activities: Inform about rejection and Inform about acceptance must be the same as that who started the first activity. Such assignment may be easily expressed in BPQL using information about the performer of the start activity and referring to a given process instance. Both kinds of information may be accessible in BPQL via built-in functions and context dependent functions. To get the performer of the first activity we may define the following BPQL query:

```
StartActivityInst(ThisProcessInst).PerformedBy.Performer.Represents.Resource
```

Selection of the order owner is the second requirement for workflow participant assignment. As was stated earlier, the owner is defined as the person who made the final acceptance of the order. For orders which value is greater that 30k EUR the owner is represented by the assigned senior seller. For the cheaper orders the owner is represented by the assigned seller. To define the owner there are at least two approaches in BPQL. The first one is based on referring to the previous activities of the currently processed activity. In this approach we may refer to the final acceptance activity as the 4th predecessor activity (instance) of the Complete documents activity. In BPQL it could be expressed as follows:

```
PredActivityInst(PredActivityInst(PredActivityInst(PredActivityInst(ThisActivityInst)))).
        PerformedBy.Performer.Represents.Resource;
```

The other solution is to include the condition on the order value directly in the owner assignment and identify the activities using their identifiers which are presented on the process model (see Figure 5.3):

```
(
 if
   for some ((ThisProcessInst.Has.ContainerAttribute where name = 'Order.Value') as A)
        ((int)(A.value) > 30000)
 then
  (ThisProcessInst.ConsistsOf.ActivityInst where id = '4')
 else
  (ThisProcessInst.ConsistsOf.ActivityInst where id = '13')
).PerformedBy.Performer.Represents.Resource;
```

The last requirement for workflow participant assignment is to select the seller with minimal current work-load. A seller with the minimal work-load is defined as a person employed at the 'Seller' position having the minimal number of tasks currently assigned. To satisfy this requirement first we have to find a bag of employees which are employed at the 'Seller' position and then check their current work-load, that is the number of the activity instances performed by them (i.e. remaining in *Created* or *Running* superstate). To define such requirement we may define the following BPQL query:

```
Resource where (position = 'Seller' and
        count(IsPerformer.Performer.Performs.ActivityInst  where
                (currState.opSuperState) in ('Created', 'Running'))
        =
        min( (Resource where position = 'Seller').
                count(IsPerformer.Performer.Performs.ActivityInst  where
                        (currState.opSuperState) in ('Created', 'Running'))
        )
)
```

5.7.3 Process Improvement Measured by Key Performance Indicators

On of the most popular technique to measuring business processes is to use key performance indicators (KPIs) [Hedge02, Junginger04]. These indicators, if chosen carefully, give an opportunity to efficiently monitoring business processes. On the basis of KPIs' values organizations are able to calculate their current and forecast their future profits/expenses, as well as plan the actions for further improvement.

An example of using KPIs in the area of workflow management is to calculate the process improvement. Such improvement is evaluated as the ratio amongst KPIs gathered for individual versions of the process. Usually the ratio is determined between the as-process and to-be processes and it is used to show what is the benefit/improvement of applying the new version of the process in a given organisation. Individual KPIs have to be defined in terms of the workflow entities included in the metamodel. Usually they operate on aggregated data.

Also for process for ordering laptops we may define a key process indicator: the value of the process. This value is defined as the weighted values of the process total cost and the overall order processing time. The total cost is defined as the average cost for a given process definition. It is represented in euros. The overall processing time is defined as the average time spend on execution of all instances of a given process definition. It is represented in milliseconds.

```
Process_value := 0.057 * avg_time + 59 * avg_cost
```

In addition, the improvement factor is defined as the ratio between the process values for the new process definition (let assume that is the version 7) and the old process definition (i.e. version 6). Let also assume that the process name is 'LaptopOrdering'.

On the basis of the above definitions and initial data, the improvement factor may be defined in BPQL in the following way:

```
sum(((ProcessDef where version = 7 and name = 'LaptopOrdering') as NewProcess).
 (0.057 * (
          sum(ProcessWaitingTime(NewProcess).resWaitingTime) +
          sum(ProcessWorkingTime(NewProcess). resWorkingTime)
          ) +
  59 * sum(ProcessCost(NewProcess).resCost)
 )) /
sum(((ProcessDef where version = 6 and name = 'LaptopOrdering') as OldProcess).
 (0.057 * (
          sum(ProcessWaitingTime(OldProcess). resWaitingTime) +
          sum(ProcessWorkingTime(OldProcess). resWorkingTime)
          ) +
  59 * sum(ProcessCost(OldProcess).resCost)
 ))
```

To calculate the improvement factor we used three built-in workflow monitoring functions, namely: *ProcessWaitingTime*, *ProcessWorkingTime*, and *ProcessCost*. Since these functions return a bag (in our case a one-element bag) we had to convert the bag to a float value using an aggregate function – sum. In addition the overall time spent for execution of all instances of a given process (definition) is calculated as the sum of the overall waiting and working times.

5.7.4 Process Verification - Reachability Property

The correctness, effectiveness, and efficiency of the business processes supported by the WfM system are vital to the organisation. Anomalies in a real workflow process usually result in runtime errors which need to be repaired on-the-fly at high cost. In order to avoid such errors we need to analyse workflow process definitions and verify their correctness [Aalst98].

One of the basic requirements for correctness of a workflow process is to check the *reachability* property. A given process definition has this property if all its activities may be reached form the start activity (i.e. every activity is either the start activity or a successor, direct or indirect, of the start activity – see appropriate definitions included in section 5.1.1.3)[9].

Reachabilty of a workflow process definition may be easily defined using BPQL and *SuccActivity* built-in function:

[9] If more than one start activity is allowed, then every activity included in the process definition has to be reachable from one of the start actvites.

```
(
  ((ProcessDef where name = 'LaptopOrdering') group as pd)
   join
  ((StartActivity(pd).resStartActivity) group as sa)
).
(count(pd.ConsistsOf.Activity) = count( sa union SuccActivity(sa, 'all')))
```

The query returns a one-element bag with a Boolean value. If the value is *true* – the property is hold, otherwise it does not hold.

To compare, because of the XML nature of XPDL the algorithm of checking this property in XPDL seems to be less readable than BPQL and away from the metamodel representation. In addition it requires some additional constructs to process nodes returned by the XPath queries[10]:

1. Find the start activity
   a) get the list of the identifiers of all activities
      XPath query: /Package/WorkflowProcesses/WorkflowProcess/Activitites/Activity@Id
   b) get the list of the activities that have ingoing transitions
      XPath query: /Package/WorkflowProcesses/WorkflowProcess/Transitions/Transition@To
   c) find an activity extracted in a) which does not belong to b)– this is the start activity (nodes processing)
2. Find all activities (their identifiers) which are the direct successors of the start activity
   XPath: query: /Package/WorkflowProcesses/WorkflowProcess/Transitions/Transition[@From=startId]@To
3. For every activity found in step 2 find all its direct successors. Repeat until all results of step 3 returns no nodes.

5.7.5 Detection of Process Anomalies

At the process design phase we usually assume that the process will work properly. Unfortunately our reality is much more rich and sometimes, especially at the first phase as the process is putting into practice, there are many failures which results in process instances termination.

In order to manage such phenomena process administrators analyse reports which shows what are the process anomalies, for instance what activities have been terminated most frequently.

Also for the process of ordering laptops we are able to generate such report. To define it we use the reliability function called for every activity included in our process. Then we sort the results starting from the lowest reliability. For two activities with the same reliability we sort them starting from the activity with the lower identifier. In addition, we also assume that we generate this report for the 7th version of the process named 'LaptopOrdering'. The report may be defined in BPQL in the following way:

```
((ProcessDef where version = 7 and name = 'LaptopOrdering') as MyProcess).
 (ActivityReliability(MyProcess.ConsistsOf.Activity) order by (resReliability, resActivity.id)).
  (resActivity.id, resReliability)
```

[10] We assume that in the analysed XPDL there is only one definition for the process of laptops ordering.

6 BPQL Applications

There are many possible ways how BPQL may be applied to support workflow management systems. Since BPQL is able to operate on the workflow enactment data it may be used to define workflow diagnosis algorithms such as identification of the process bottlenecks or detection of the process activities susceptible to exceptional situations. Such information may be then used to improve business processes. BPQL may also be used to verify correctness of workflow processes as it was presented in the previous chapter for reliability property.

It seems that the most demanding and spectacular application of BPQL, however, would be enriching a process definition language to make it more flexible. In this case, BPQL would enable process designers to express at the process definition phase some constraints and rules which depend on process execution history (past), current application relevant data as well as workflow control data (present), and some forecasts of further process executions (future). This feature is especially important for less rigid processes such as administration ones which need to adapt frequent changes in workflow environment as well as workflow itself. An example of such need is the case when it is required to select the workflow participant with the minimal current workload.

This chapter presents the approach how to integrate BPQL into a standard business process definition language and how to extend the WfM system architecture by the BPQL manager. In addition, it also describes how this extended architecture has been prototypically implemented in a commercial system, namely OfficeObjects® WorkFlow. This work was partially done within the European research project ICONS [ICONS02]. Finally, it also portrays a real example of the BPQL usage in the European Exchange of Documents-Poland system (EWD-P system).

6.1 Integration of BPQL into XPDL

It seems that the best way of using BPQL in process definition is to integrate it with a well known and widely used process definition language. Nowadays, there are several standard process definition languages such as XML Process Definition Language [WfMC-TC-1025], Business Process Modelling Language [BPML02] or more web-oriented languages such as Business Process Execution Language for Web Services [BPEL4WS03] and Web Service Description Language [WSDL01]. All these languages use XML format.

XML Process Definition Language (XPDL) is promoted by Workflow Management Coalition as an XML successor of Workflow Process Definition Language. It is conceived as a graph-structured language with additional concepts to handle blocks. Scoping issues are relevant at the package and the process levels. Routing is handled by specification of routing operators (i.e. split and join) and transitions between activities. Conditions associated with transitions determine

at execution time which activity (activities) should be executed next. XPDL focuses on issues relevant to the distribution of work. An activity requires a resource to perform it and an application to implement it.

Business Process Modelling Language (BPML) is promoted by the Business Process Management Initiative (BPMI) and is based on well established and comprehensive concepts of workflow management defined within the WfM coalition's workflow standard framework. It is conceived as a block-structured programming language. Recursive block structure plays a significant role in scoping issues that are relevant for process modelling, definition and enactment. Flow control is handled entirely by block structure concepts. BPML focuses on issues important in defining web services. This is reflected in providing a set of basic activity types such as loop, switch, receive an event, as well as supporting exception handling and compensation mechanisms.

Business Process Execution Language (BPEL) is promoted by IBM and Microsoft. It is a block-structured programming language allowing recursive blocks but restricting modelling and definition to the top level. Within a block graph-structured flow concepts are supported to a limited extent, constrained by inheritance from previous generation workflow software (only acyclic graphs, hence no loops; some constraints on going across block boundaries; a complicated semantics for determining whether an activity actually happens). BPEL focuses on issues important in defining web services and does that in a way which is quite similar to BPML.

Web Services Description Language (WSDL) is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate.

So far it seems that XPDL is the most mature and complete process definition language and may be easily extended with BPQL. This integration will much extend the functionality of the existing language.

6.1.1 The approach

As was stated earlier, BPQL may be used to make process definition more flexible. Especially, BPQL may be applied for those process definition elements which have to be evaluated at the run time querying application relevant data or workflow control data.

BPQL may be useful in definition of process flow conditions, that is transition conditions as well as pre and post activity conditions. Usually in typical flow conditions, BPQL queries will express dependencies on data container, while for more complex conditions, they may represent intricate operations on some elements of data container or verification of the QoS constraints.

BPQL is also essential for Workflow Participant Assignments (WPAs). It helps defining such WPAs where it is needed to refer to users already involved in the process execution or to find the optimal performers according to the various criteria such as the number of tasks assigned, total time to complete or the performance cost.

In the next sub-sections we present how the mentioned XPDL process definition elements may be adjusted in order to represent BPQL queries. Since pre and post conditions are not represented in XPDL directly we propose a way how they may be added to the activity definition using the *ExtendedAttribute* XML tag.

Moreover, in the XPDL+BPQL integration we will use the three basic rules:

- The XPDL schema[11] must remain the same as for the original XPDL. Such a rule allows XPDL+BPQL process definitions to be imported into XPDL compliant WfM systems and vice versa. In consequence, the effort to adapt existing WfM systems and XPDL process definitions to the extended language will be drastically reduced.
- The standard XPDL elements[12] are preferred. The extended attributes are used only if there is no way to express required constructs by the standard XPDL elements. This rule reduce the effort to adapt semantically existing XPDL process definitions.
- The XPDL+BPQL extensions must be as simple as possible. The current trend in generation of hundreds pages of documentation seems to be unrealistic for understanding by many potential users.

6.1.2 BPQL as the XPDL scripting language

XPDL makes it possible to use an external language to define XPDL expressions. In XPDL terms it is called a scripting language. Specification of this language is given by *script element* which is a part of the package definition. This element includes three attributes:

- Type – this identifies the scripting language. For BPQL we recommend to use the value: *text/BPQL*.
- Version – this is the version of the scripting language. For BPQL the value should be set to *0.5*.
- Grammar – this is a reference to a document that specifies the grammar of the language. For BPQL it is the reference to the BNF definition of the BPQL syntax.

An example of the XPDL definition of this element is given below:

```
<Script Type=”text/BPQL” Version=”0.5” Grammar=http://icons.rodan.pl/BPQLGrammar.bnf />
```

[11] XPDL schema is an XML schema defined for the purpose of XPDL by WfM Coalition (see http://www.wfmc.org/standards/docs/TC-1025_schema_10_xpdl.xsd).

[12] XPDL element is an XML tag defined within XPDL schema.

6.1.3 Transition Condition

In XPDL a transition condition is expressed by the *Transition/Condition* element. This element includes one attribute: *Type* and one sub-element: *Xpression*. The attribute represents the type of condition. Usually it is set to CONDITION. The sub-element represents the condition written in the scripting language (that defined above in the Script element). In addition, the condition itself is also represented as the value of the Condition element.

For BPQL queries we recommend to write a given transition condition in both places: as the value of the condition element as well as the value of *Xpression* element. This dualism on the one hand makes possible to use XPDL process definition for those WfM systems which do not use Xpression element and, on the other hand, allows WfM systems to verify such conditions against BPQL syntax as they do for scripting XPDL expressions. An example of the BPQL transition condition element written in XPDL may look like:

```
<Transition Id="b1" From="ChckBalance" To="ProcRequest">
   <Condition Type="CONDITION">
     for some ((ThisProcessInst.Has.ContainerAttribute where name = 'Order.Value') as A)
                    ((int) (A.value) > 30000)
     <Xpression>
       for some ((ThisProcessInst.Has.ContainerAttribute where name = 'Order.Value') as A)
                    ((int) (A.value) > 30000)
     </Xpression>
   </Condition>
</Transition>
```

6.1.4 Workflow Participant Assignment

According to the WfMC's definition, a workflow participant assignment defines the set of participants that will perform a given activity. In XPDL two XPDL schema elements are used to define workflow participant assignments. The first one is the *Participant* element. This element specifies all the workflow participants which may perform any of the activities defined within a given process. A participant element possesses a set of standard attributes such as *id* and *name* as well as user-specific extended attributes. The second element is the *Performer* element. Its value specifies the (potential) performer of a given activity. Usually it refers to a workflow participant. This element has no attribute.

To be coherent with the above definitions it is suggested to use BPQL for defining a role (i.e. a type of workflow participant). In that case a BPQL definition could be included as an extended attribute of the role (participant) specification. A BPQL query would return a set of workflow participants that would satisfy it. In addition, also WPA decision and the modifier

introduced in [Momotko02a] could be used to specify a workflow participant. An example written in XPDL may look like:

```
<!-- Workflow Participant definition -->
<Participant Id="p1" Name=”Seller”>
   <ParticipantType Type="ROLE">
   <Description>Seller</Description>
   <ExtendedAttributes>
    <ExtendedAttribute Name=”Definition”>
        Resource where (position = ‘Seller’)
    </ExtendedAttribute>
  </ExtendedAttributes>
</Participant>
. . .
<!-- and workflow participant assignement specification – performer -->
<Activity Id =”1” Name=”Enter Order”>
  . . .
  <Performer>p1</Performer>
</Activity>
```

6.1.5 Pre and Post Activity Conditions

As stated earlier, pre and post conditions for workflow activities are not defined directly in XPDL. However, it is possible to use the *ExtendedAttribute* XPDL schema element of the *Activity* element in order to express these conditions which, once again, would be defined in BPQL. We recommend to use the names: *PreCondition* and *postCondition* in the ExtendedAttribute element for pre and post conditions respectively. An example written in XPDL may look like:

```
<Activity Id="56" Name="Compose Acceptance Message">
  <Implementation>
   <Tool Id="composeMessage" Type="APPLICATION">
    <ActualParameters>
     <ActualParameter>status</ActualParameter>
    </ActualParameters>
   </Tool>
  </Implementation>
  <ExtendedAttributes>
    <ExtendedAttribute Name="PreCondition">
      ProcessInstCost(ThisProcessInst) < 321
    </ExtendedAttribute>
  </ExtendedAttributes>
</Activity>
```

6.2 The Integrated WfM + BPQL Architecture

In the previous sections we described how BPQL may be integrated with a process definition. As stated earlier, BPQL queries may be used to define by process designers some elements of workflow process definition such as workflow participant assignment and process flow conditions. Such defined queries are then evaluated during process enactment.

This section details the above scenarios and specifies the required BPQL functionality. It also presents the architecture of an integrated WfM + BPQL system which is able to provide the required functionality. This architecture shows the dependencies between the BPQL manager and the rest of the WfM system. These dependencies are then used to describe more formally the required and provided interfaces of BPQL Manager.

At design time, a process designer defines a workflow process and, if required, specifies BPQL queries. To build these queries information about BPQL operators, functions and other constructs is used. This information is provided by the BPQL Manager. At any time process designers may verify a defined BPQL query. On the user request, the BPQL Manager checks whether such query is syntactically correct. If it is not, a design time exception is thrown. Process designers may also add a new built-in BPQL function. Such user-defined functions extend the BPQL functionality and behave as the standard built-in functions. They are registered in the BPQL repository and may be used in any BPQL query. If a built-in function was improperly introduced and is not currently used, the process designer may also remove such function.

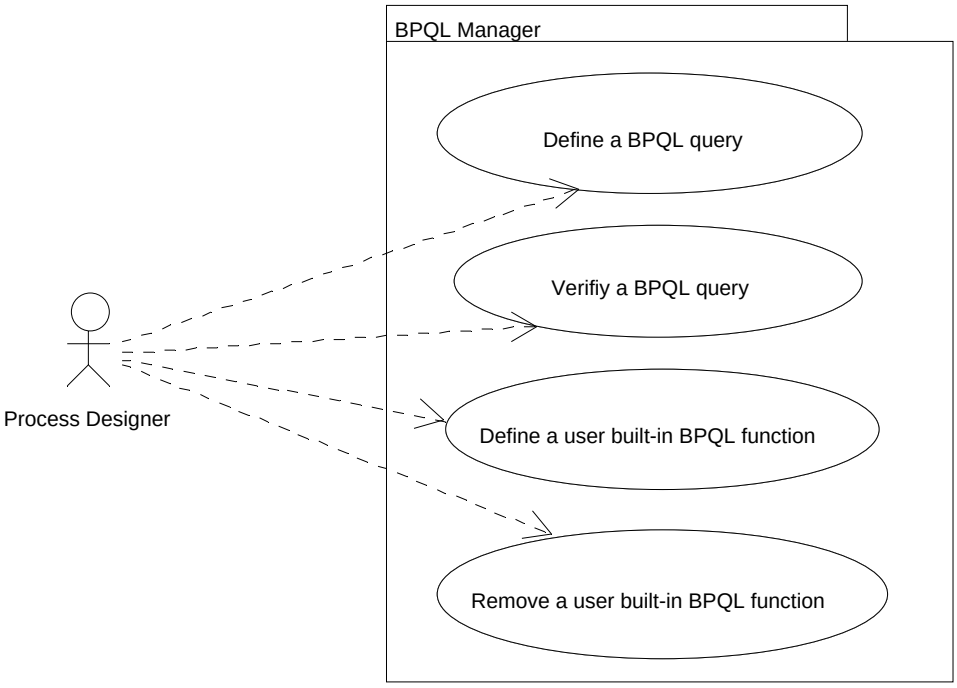

Figure 6.1 The design time use cases for BPQL Manager

At run time, workflow participants execute process instances. When they change the state of a process instance (or their activity instances) the WfM system evaluates, if required, BPQL queries (e.g. to check if a transition condition is satisfied or to evaluate the workflow participants who will perform a given activity). At this stage the BPQL Manager provides a function to evaluate a BPQL query and returns its result. The evaluation result is usually a collection of references to objects. If the evaluation process fails, a run time exception is thrown.

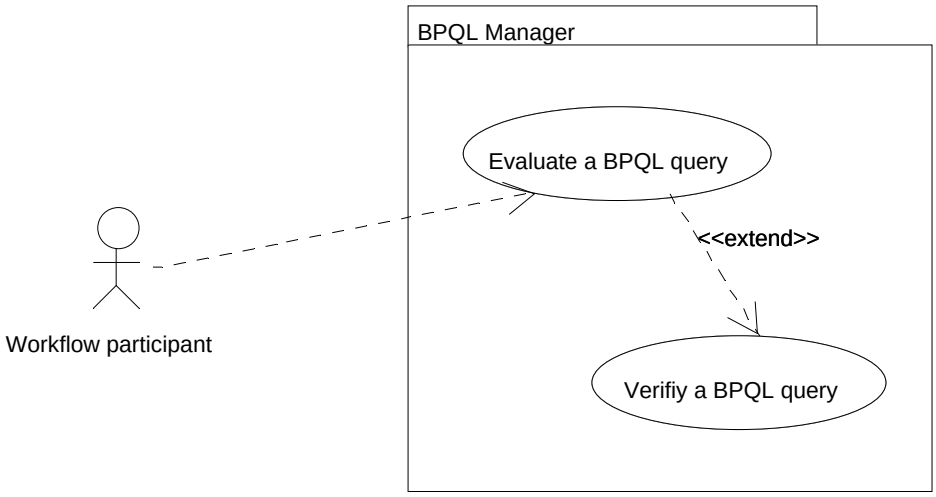

Figure 6.2 The run time use cases for BPQL Manager

Summing up, the BPQL Manager has to:

- provide information about the BPQL operators, built-in functions and other constructs
- enable a new built-in function to be added to the BPQL repository
- enable an existing built-in function to be removed from the BPQL repository
- verify if a given BPQL query is correct. Possible errors in the query definition (design time errors) are reported by throwing an exception
- evaluate a BPQL query and returns its result. Possible errors in the query evaluation (run time errors) are reported by throwing an exception.

The first three functionalities are used at the process design time while the next two functionalities are exploited at process run time. In addition, the BPQL manager needs information of :

- workflow relevant data (i.e. document attributes stored in data container) which may be used in process control flow conditions
- organisational data (e.g. users, groups, roles) which may be used in workflow participant assignment
- workflow control data (e.g. process instance, activity instance) which may be used in any element of process definition.

The architecture of an integrated WfM+BPQL system providing the described functionality and operating on the enumerated information is presented below. This architecture assumes that there is an IT system which includes three sub-systems:

- Repository – to manage application data. A part of these data is used by WfM system to define process flow conditions (workflow relevant data).
- Resource Management System – to manage organizational data which are used to define workflow participants
- Workflow Management System – to define, enact and monitor workflow processes.

A part of WfM system is the BPQL Manager. This manager is responsible for definition and evaluation of BPQL queries. It operates on workflow objects stored in the Workflow Repository. The BPQL Manager provides its functionality to two other WfM modules, namely the Process Designer and Workflow Enactment engine. The Process Designer is responsible for modelling and defining workflow processes. It requires the following functionality from the BPQL Manager:

- Search and retrieve a list of BPQL elements that is functions, operators or other BPQL constructs. It is possible to set a condition on the element selection. This condition is a BPQL query which may include attributes of the BPQL elements, Boolean operators and comparison operators.
- Create a new BPQL built-in function. First the Process Designer specifies all required function attributes such as name, result type and then sends a request to the BPQL Manager to add the function to the BPQL repository.
- Remove a user-defined function which already exists in the BPQL repository. To do that, the Process Designer has to know the identifier of the function.
- Verify a BPQL query. The BPQL Manager should be able to provide information on possible errors which may exist in the query.

The Workflow Enactment engine is responsible for enactment of workflow processes. It requires from the BPQL Manager the following functionality:

- Evaluate and verify BPQL query. The evaluation result should be returned as a collection of objects. The BPQL Manager should be able to provide information on possible errors which may exist in the query.

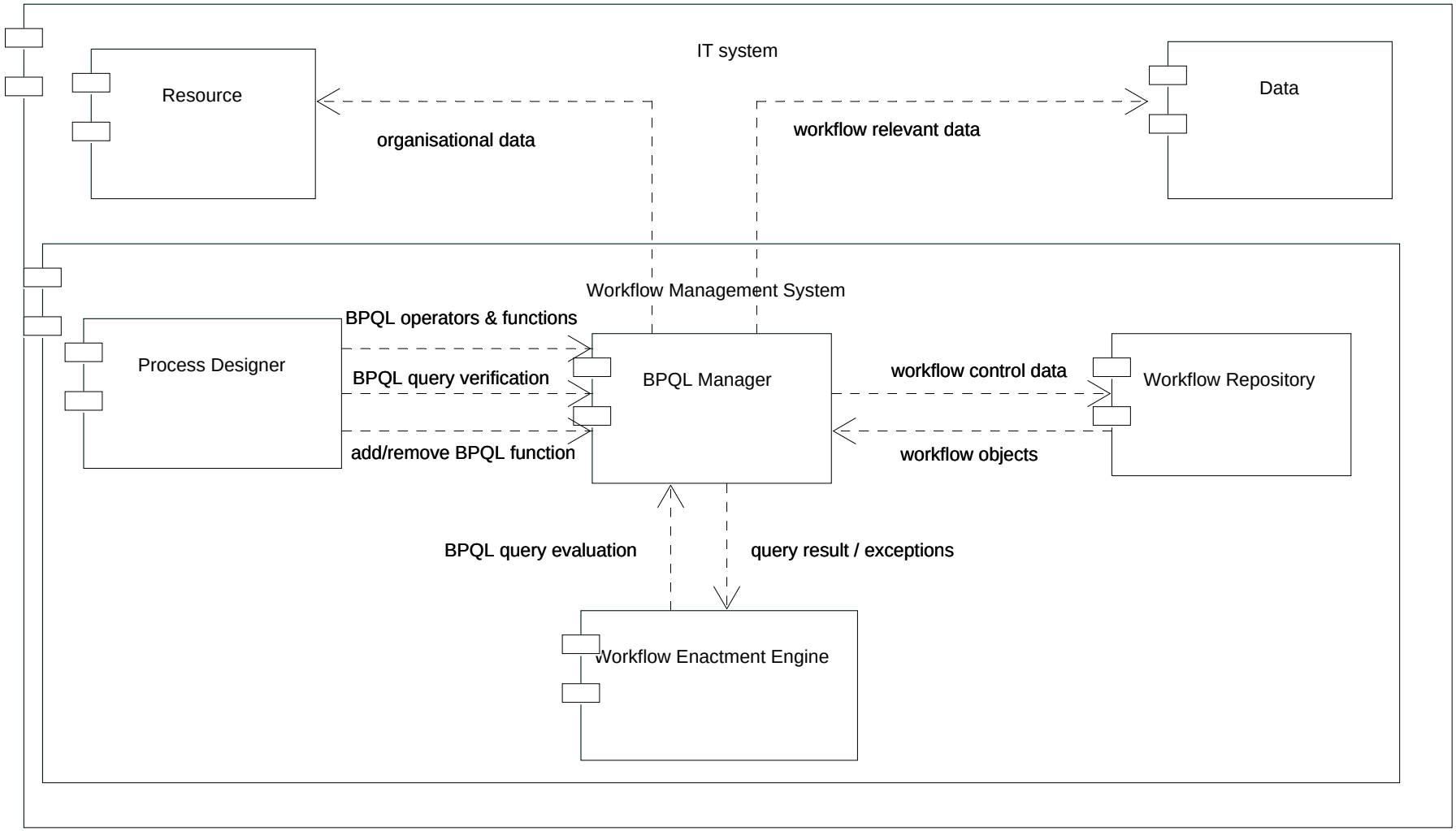

Figure 6.3 The WfM + BPQL functional architecture

6.3 Detailed BPQL Architecture

In the next stage the architecture of the integrated WfM+BPQL system needs to be detailed to identify the main components of the BPQL Manager and to specify its interfaces in terms of classes and interfaces provided.

There are three basic components in the BPQL Manager:

- **BPQL Parser** responsible for syntactic and (partially) semantics analysis of BPQL queries. It takes a BPQL query as the input and generates its syntactic tree. During parsing it uses the BPQL grammar and information about the operators, functions and other BPQL constructs. Both kinds of information are stored in the BPQL Repository. If there is an error during parsing, it throws a BPQL exception.
- **BPQL Evaluator**. On the basis of the syntactic tree for a BPQL query, the BPQL Evaluator evaluates it. As the result of the query the BPQL Evaluator returns a collection of values which satisfies the query. During evaluation it uses data stored in the Environmental and Query Result Stacks. These data are managed by the BPQL Repository. In this evaluation, the BPQL Evaluator may also use workflow objects, workflow relevant data and organizational data. If there is an evaluation error, it throws a BPQL exception.
- **BPQL Repository** responsible for storing BPQL built-in operators and monitoring functions, the current values of ENVS and QRES stacks as well as the query evaluation context and other BPQL auxiliary data.

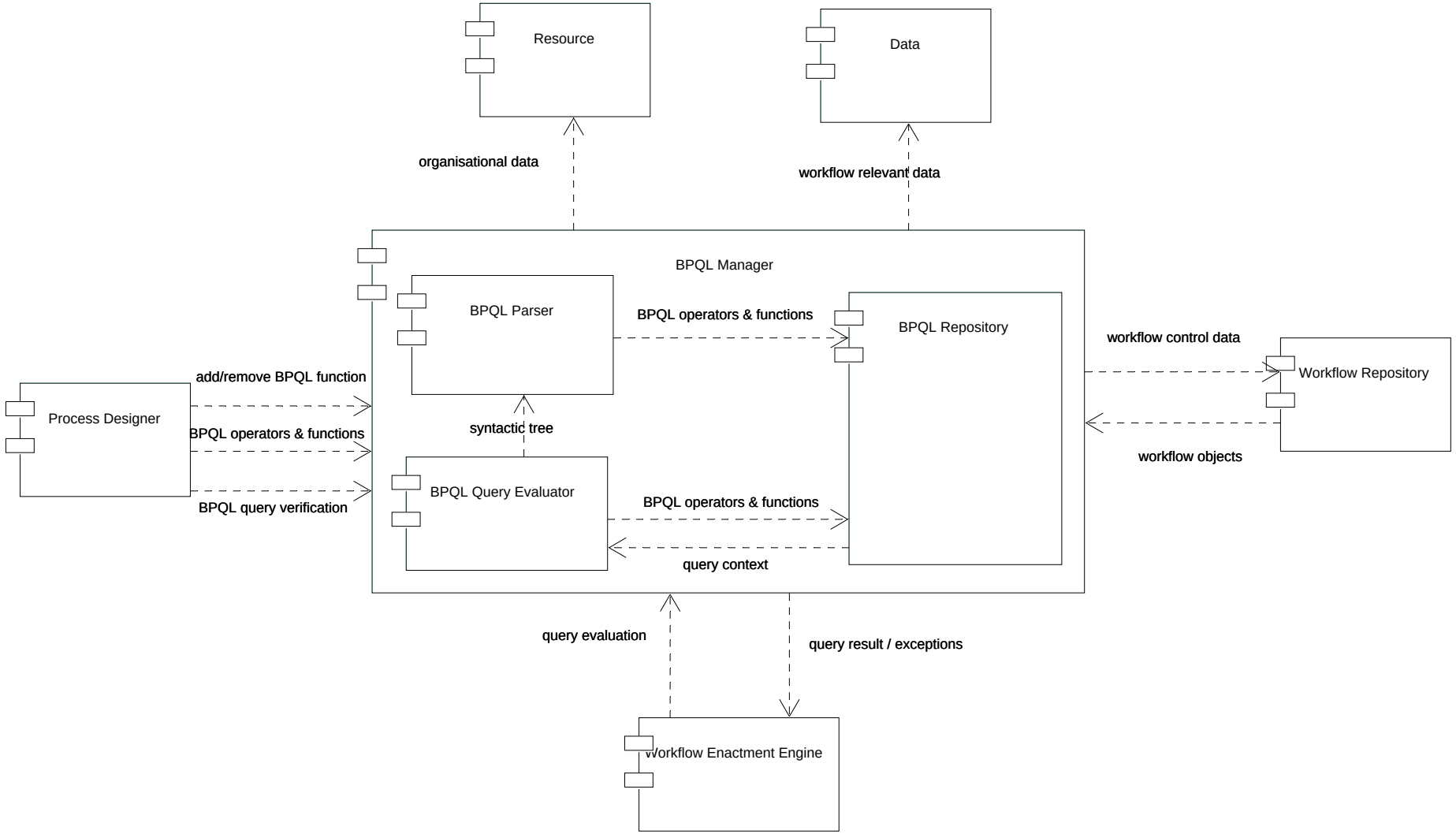

Figure 6.4 The detailed architecture of the BPQL Manager

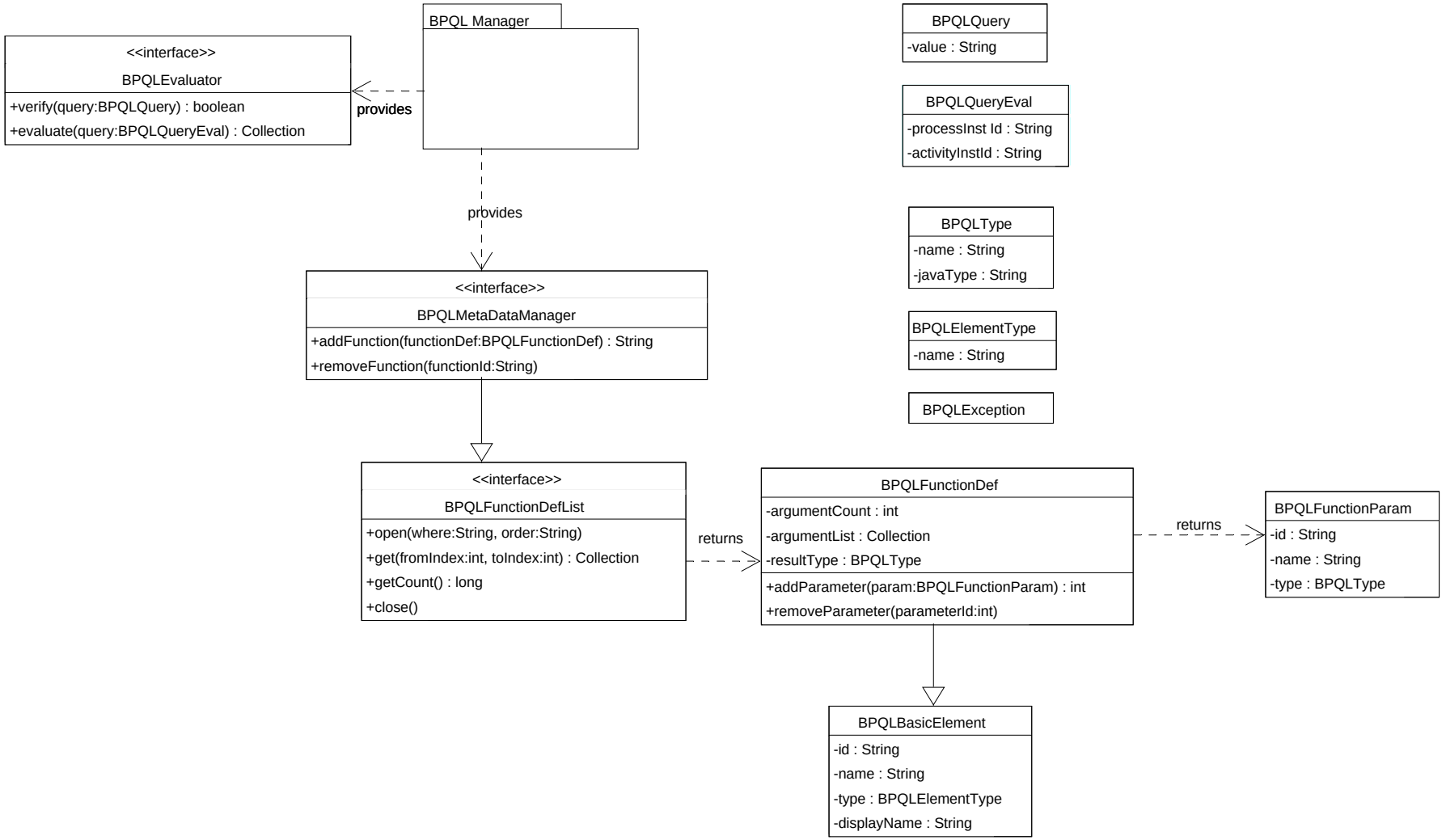

Figure 6.5 The model of the provided interfaces

6.3.1 Provided Interfaces

The BPQL Manager provides to the WfM system a set of interfaces (functions) and classes (only Java beans). The BPQL Manager provides two main interfaces: *BPQL MetaData Manager* and *BPQL Evaluator*. The former specifies functions to retrieve design time information while the latter has two functions to verify and evaluate BPQL queries during run time. The detailed specification of these interfaces and its elements is given in Appendix C.

6.4 Implementation in OfficeObjects®

The first implementation of the Business Process Query Language was done by the Rodan Systems (www.rodan.pl), mostly within the Intelligent base CONtent management System, an FP5 EC research project [ICONS04]. One of the main goals of this project was to define and develop flexible mechanisms and tools to represent procedural knowledge. In order to achieve this goal, the ICONS project proposed to make existing WfM systems more flexible by introducing:

- **Flexibility in process definition** – it extended process definition by dynamic constraints and rules (instead of static ones present in the existing WfM systems). These dynamic elements are to be defined on the basic of execution history (past), current application relevant data as well as workflow control data (present), and some forecasts of further process executions (future).
- **Team collaboration management** – it enabled virtual teams to collaborate within a given process or activity with a purpose of solving a specific knowledge work problem. There is no restriction on time, order and number of collaborations between the team members. The members of such team are either workflow participants of a given process instance (process collaboration) or multiple performers of a given activity (activity collaboration).
- **Awareness of selected QoS parameters** – it applied an advanced time management mechanism based on the ePERT method [Eder99]. This mechanism gave an opportunity to monitor durations and deadlines at both process and activity levels as well as to detect if a given time constraint violation is critical for the whole process.

To assure flexibility in process definition and to express dynamic constraints and rules, BPQL was used. Its first prototypical implementation has been done within OfficeObjects® platform and is known as OfficeObjects®BPQL (OO BPQL). Within this implementation we developed BPQL process flow functions and workload functions. OO BPQL was integrated in the OfficeObjects®WorkFlow system, extending XPDL process definition of flexible workflow participant assignment as it is described in section 6.1.4.

OfficeObjects®WorkFlow is an embedded workflow management system which supports the definition, enactment, and monitoring of business processes in accordance with appropriate corporate standards [Momotko01, 02b, 03a, 03b]. It guarantees that each activity specified in the system will be carried out according to the applicable procedure and each subsequent stage of the scheduled task will be completed by an indicated person, in a specific timeframe and with the use of specific data. OfficeObjects®WorkFlow eliminates accidental and inconsistent achievement of set goals and motivates the managing persons to introduce an ordered organizational structure. The advantage of OfficeObjects®WorkFlow is that it uses tools which support system operation, including graphic tools for planning the workflow, an extensive administrator's panel for system management and advanced mechanisms for archiving all information concerning executed tasks.

OfficeObjects®WorkFlow is regarded as a "second generation" WfM system with well defined semantics provided a comprehensive workflow ontology. It is also compliant with the famous workflow management standards. The workflow process metamodel is an extension of that proposed by the Workflow Management Coalition [WfMC-TC-1011]. The process model is expressed in Business Process Modelling Notation [BPMN04] proposed by Business Process Management Initiative (BPMI). The process definition is expressed in XPDL and visualization of process execution is compliant with BPMN extension proposed in [Momotko03].

The OO BPQL architecture is compliant with that one presented in section 6.3 . Its conceptual design and detailed design were mainly expressed by UML diagrams (version 2.0). Its functional architecture and interfaces were described using UML component diagrams. Detailed class design was defined using UML class diagrams. OO BPQL behaviours were expressed in terms of the sequence and activity UML diagrams. In addition, OO BPQL database design was done as an ERD diagrams.

The BPQL Manager, the heart of OO BPQL, was implemented in Java (version 1.4) as a Java package. Its interfaces are provided as Java interfaces and Java beans. The BPQL Parser has been based on JavaCC parser generator which generates a top-down (recursive descent) parsers as opposed to bottom-up parsers generated by YACC-like tools. By default JavaCC generates LL(1) parsers. The main part of BPQL Repository (i.e. BPQL meta data) is represented by relational structures in an RDBMS. The remaining part is stored directly in main memory (i.e. QRES, ENVS).

6.5 The Polish Government – European Commission Interoperability

The results of the ICONS project, especially OfficeObjects® WorkFlow extended by OO BPQL features has been applied in the European Exchange of Documents – Poland system (EWD-P

system). This section describes the EWD-P system focusing on the way how OO BPQL helped deploying the system successfully.

6.5.1 The EWD-P system Application Domain

The documents from the General Secretary of the Council of the European Union (referred further to as UE documents) are received from the U32Mail system a (legacy system exploited also by other EU countries). An EU document consists of its envelope and a set of enclosed files of various types (PDF, HTML, Doc97). The envelope comprises of a set of attributes that characterise the nature, source and destination of the document. The files constitute the real content of the document. So far, Poland receives the EU documents in either English or French. Based on EU subject codes and other attributes included in the document envelope as well as in the document content, the EWD-P system classifies it and selects the most corresponding Polish subjects which this document refers to. In the next stages, the selected Polish subjects are used to determine organisational units and experts that are responsible for processing of a given document.

For the majority of the EU documents a Polish response must be prepared and delivered by a certain date. If the deadline for preparation is not met, it is assumed that there is no objection to the original EU document and that the document is fully accepted. When the Polish response is prepared, it is sent to the Polish Representation Office at Brussels which in turn presents the document to the Council's pertinent teams.

From the information management point of view the 'case' concept plays the crucial role as it secures all information related to a given response. This comprises the original EU document (triggering the corresponding case; perhaps in versions), response, related cases, external documents etc. The cases are maintained by cooperating experts who attach the consecutive versions of the Polish response as well as other relevant information (e.g. legislative documents). The cases store minutes of meetings on which Polish representatives present the Polish response (this includes remarks and comments).

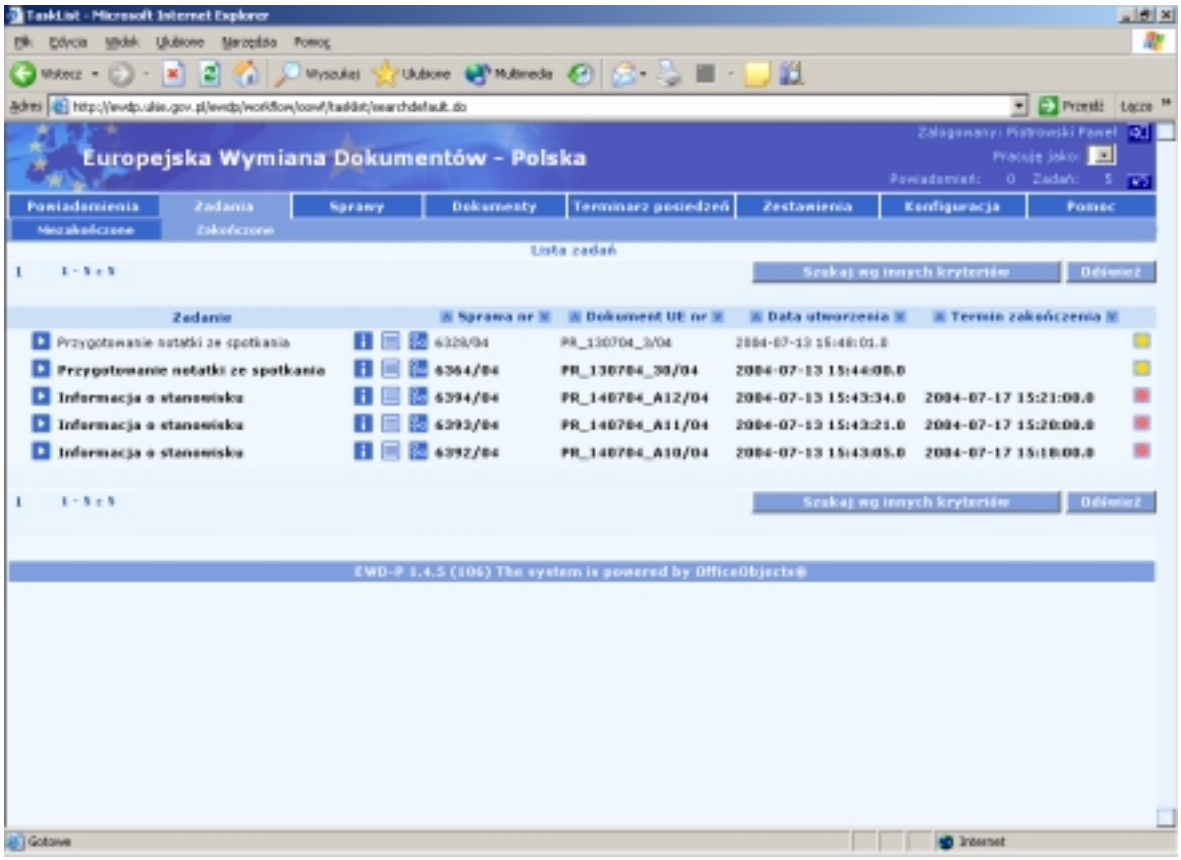

Figure 6.6 An example of the EWD-P work list (in Polish)

It is critical to ensure consistency with other cases and, to the same extent, preserve the information on previously existing cases that somehow influenced a given response. A net of relations among cases facilitates navigation among related cases and promotes the access to global view on some issues rather than to its individual aspects.

The system dynamics is imposed by processes executed by the workflow engine – one process instance for each EU document. The workflow starts from publication of an original document on the U32Mail gateway, continues through initial dispatching in the Committee for European Integration, precisely dispatching on the level of particular ministries' units, establishing collaboration among the selected experts, elaborating of final standpoint and ends up with the final delivery to the Polish Representation Office at Brussels (Figure 6.7). The Polish response preparation activities are supervised by the leading expert. At the end of this stage, all involved experts vote if they agree on the final Polish standpoint.

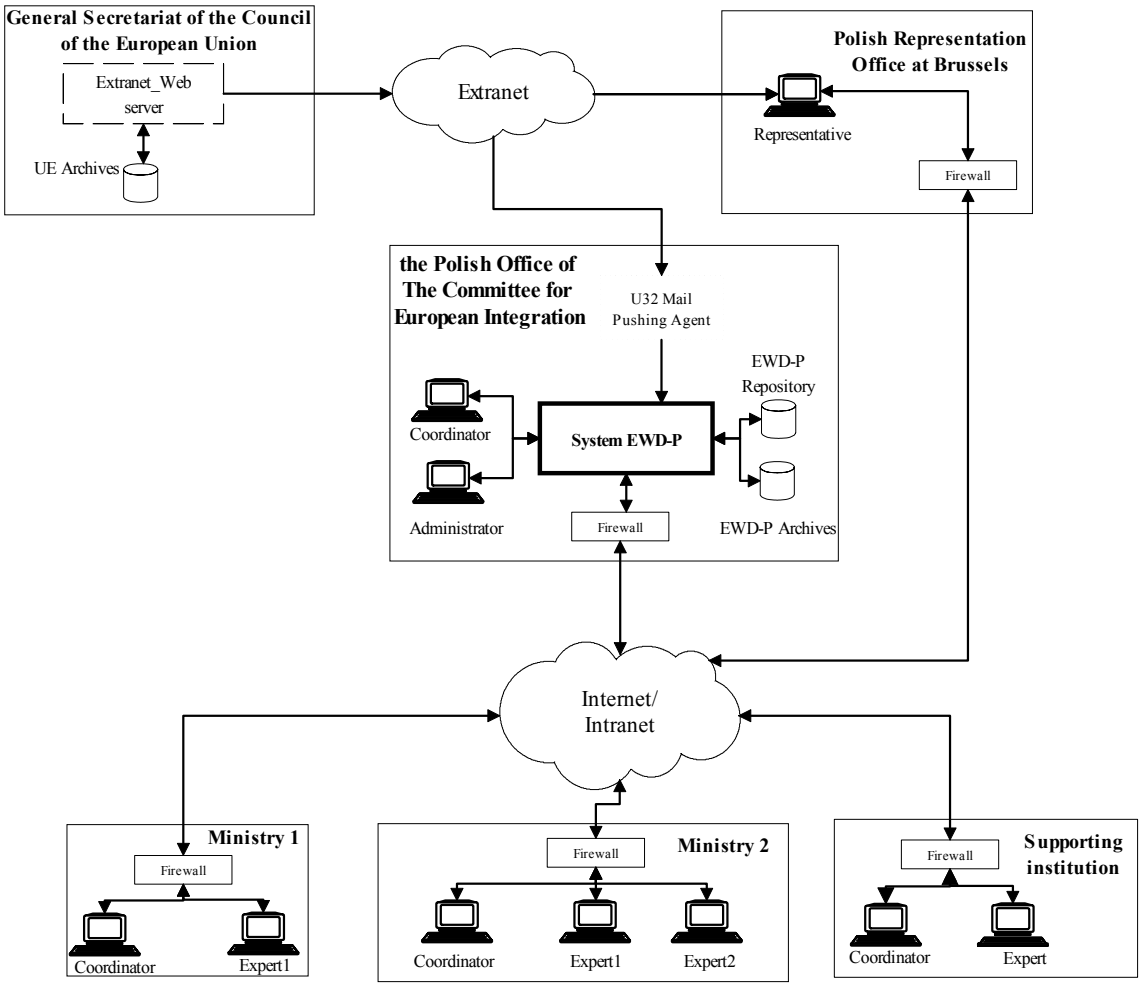

Figure 6.7 Actors of the EWD-P architecture

Besides its mission-critical functionalities the EWD-P system serves a number of auxiliary services. First of all, the users can search over the repository of documents and cases using attribute-based search (supported by user friendly search criteria builder) and full text search. This covers also European documents repository, accessible via the U32 gateway. This fosters knowledge reuse based on previous experiences. Secondly, the system provides full accountability of experts' decisions by storing the information on contributions of individuals to the documents and cases as well as their behaviours in a concrete workflow process instance (e.g. missed deadlines). Thanks to this, in the case of any affair concerning the essentials of a given standpoint, the standpoint's development history can be examined in every detail. The complete information on cases (together with specific contributions provided by individuals) are archived according to the stringent central government practices and standards and accessible for search as well as retrieval but closed for any modifications.

6.5.2 Advances in Workflow Management

The implementation of the EWD-P dynamics is fully based on the workflow management technology. The generalised workflow process (in the current version - comprising of 45 activities and involving 8 roles of executors) explicitly defined in the form of Conceptual Business Process Model and XML Process Definition Language is responsible for classification and distribution of the EU official documents as well as for preparation of the Polish response. The process of elaboration of the Polish standpoint is instantiated for every incoming EU document and then executed by the workflow management system.

The major challenge addressed by the workflow engine in the EWD-P system was to optimize workflow processes executions in order to "to assure that appropriate activities will be performed by the right (knowledgeable) participants based on pertinent information in due time". A process instance is perceived by the workflow participant through a task list in which all commitments of a given user (possibly involved in a numerous cases) are sorted along with some priorities. The key features that made the workflow engine successful within the EWD-P system are:

- dynamic workflow participant assignment based on BPQL queries,
- flexible communication among workflow process participants,
- advanced time management,
- process execution visualization.

6.5.3 Detailed Advances in OO BPQL

OO BPQL played a very important role in the EWD-P process definition. BPQL queries was used to define workflow participants assignments. These assignments were defined on the basic of workflow relevant data (EU document attributes, its subjects), workflow control data (process history) and resource information (users, organisational structure, positions, and roles).

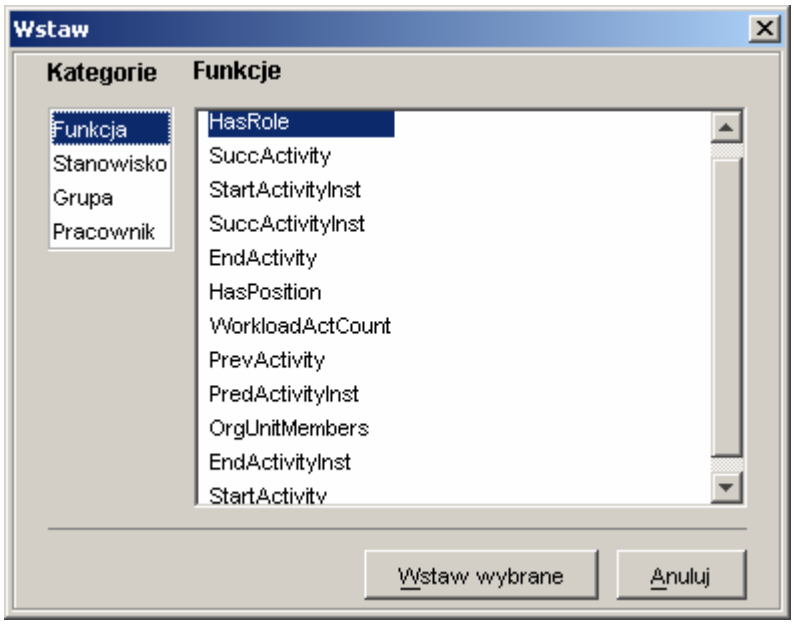

Figure 6.8 A user dialog to select buitl-in monitoring functions (in Polish)

In addition the process flow and some of the workflow participant assignment built-in monitoring functions have been applied (see Figure 6.8).

In EWD-P process the following roles were defined using BPQL queries:

- Main coordinator – a user employed at the position of the 'coordinator' at the ministry which plays the role of the main coordinator.
- Leading coordinator – a user employed at the position of the 'coordinator' at the organisational unit playing the leading role for the subject which a given EU document concerns.
- Supporting coordinator – a group of users employed at the position of the 'coordinator' at the organisational units playing the supporting role for the subject which a given EU document concerns.
- Leading expert – a user playing the 'leading expert' role for the subject which a given EU document concerns.
- Supporting experts - users playing the 'supporting expert' role for the subject which a given EU document concerns.
- Negotiator - a user employed at the position of the 'negotiator'.
- Representative – a user employed at the position of the 'representative' playing the negotiator role for the subject which a given EU document concerns. S/he represents a Polish standpoint at Brussels.

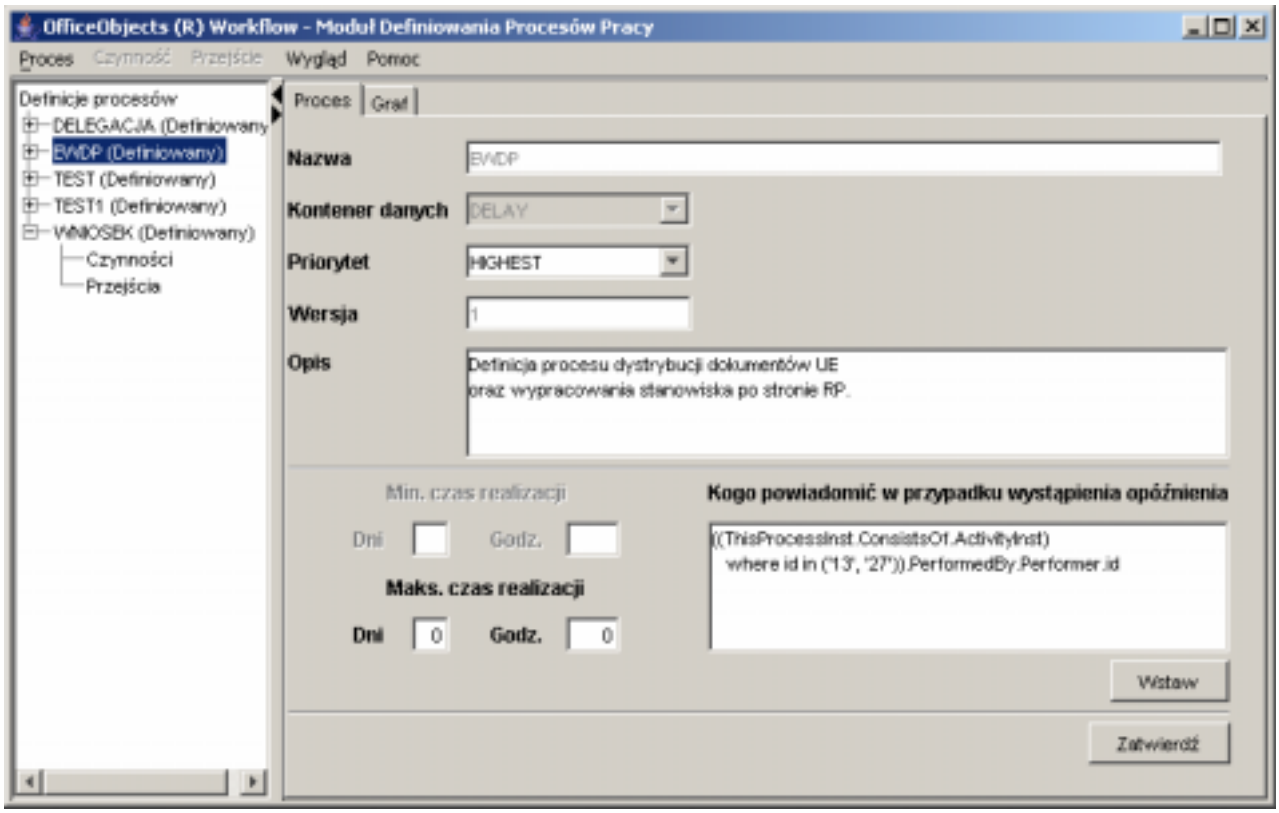

Figure 6.9 An example of how to define workflow participants who will be informed of possible time constraints violation (in Polish)

Owing to OO BPQL features in EWD-P it was possible to:

- Express complex assignments based on the EU document subjects and roles possessed by users related to these subjects It was possible to express these rules directly in the process definition rather than in the application code. That made the EWD-P easier to maintain.
- Make process definition more coherent and generalised; one for all ministries.
- Use the process context and history (i.e. workflow control data) to deliver information about the standpoint presentation results to the experts who did participate in the standpoint elaboration.
- Signalise violation of the time constraints to the right coordinators and experts (see Figure 6.9).
- Assure that the standpoint elaboration has been done by the right number of experts (i.e. one or more experts).

7 Conclusions and Future Work

There were two main goals of this work. The first one was to define a language and a set of tools (or functions) to monitor workflow processes and to use them within a standard and widely known process definition language in order to make it more flexible and able to cope with dynamic changes. The second one was to practically verify the usefulness of the proposed approach by implementing a prototype and then to apply it in a real application.

To achieve the above goals first we defined an appropriate process metamodel, developed BPQL - a language to query this model, then built workflow monitoring functions within BPQL and integrated this query language with XPDL - a standard process definition language. Finally we developed a prototype version of the language and monitoring function in OfficeObjects®WorkFlow system as OfficeObjects®BPQL [Momotko04] and verified its usefulness within EWD-P system - a real practical application.

The EWD-P application showed the practical usefulness of BPQL and its built-in workflow monitoring functions. Owing to OO BPQL features in EWD-P it was possible to:

- express complex assignments on workflow participants, especially where workflow relevant data and workflow control data were required.
- signalise violation of the time constraints to the right workflow participants.
- assure the appropriate number of workflow participants responsible for performing a given activity.

The presented approach, however, leaves several open issues. In some cases where relevant data come from several data sources it is impossible to make a BPQL query. Also BPQL is not helpful when the current control of process resources depends on factors not included in the metamodel (e.g. reliability of resources). In such cases the workflow processes must be controlled manually. So far, the effectiveness of BPQL is promising, however it still needs to be verified in more detail. In the opinion of the author, the EWD-P application may not be regarded as a sufficient evidence of the overall BPQL efficiency (i.e. in OfficeObjects®WorkFlow BPQL was used for workflow participant assignment only).

Since BPQL and its monitoring functions is a universal way to query workflow metadata, there are at least several directions of BPQL enhancements and further applications. The most important WfM sub-domains where BPQL may be applied are:

- process verification – BPQL may be used as a universal and system independent language to express algorithms and tools to verify process definitions. For example, in the section about built-in workflow monitoring functions we showed

how to define a function to check the soundness property (i.e. whether it is possible to reach every activity from one of the start activity).

- process analysis and optimisation – BPQL may be used to query process history (i.e. population of process instances) in order to find some irregularities in process execution which then may be removed to make the process definition more optimal.
- Process mining, mining social network – BPQL may express algorithms for finding various relations among workflow objects such as "who *likes* to work with JohnB", etc.
- QoS monitoring – BPQL may also be used to express QoS constraints (or QoS rules) on process/activity execution.

On the other hand, there are first symptoms that the interest in BPQL is growing. First of all Rodan has decided to develop the second version of BPQL. Comparing to the first version of OO BPQL, the new version will probably use BPQL to define transition conditions as well as pre & post activity conditions. Secondly, it is likely that BPQL will be used for monitoring of QoS factors within the Adaptive Service Grid project, phase B (see http://asg-platform.org) sponsored by the European union within the 6th Framework Programme.

As the final remark, the author encourages the readers of this document to join the group of BPQL propagators hoping that BPQL will eventually become a widely known standard for the business process query language.

References

[Aalst02] Aalst, W.M.P. van der, Dongen, B.F., van, Herbst, J., Maruster, L., Schimm, G., Weijters, A.J.M.M.: *Workflow Mining: a Survey of Issues and Approaches*, 2002.

[Aalst95] Aalst, W.M.P. van der: *A class of Petri net for modelling and analyzing business processes*, Computing Science Reports 95/26, Eindhoven University of Technology, Eindhoven, 1995.

[Aalst98] Aalst, W.M.P. van der: *The Application of Petri Nets to Workflow Management*, The Journal of Circuits, Systems, and Computers, 1998.,

[Aalst99] Aalst, W.M.P. van der: *Generic Workflow models: How to Handle Dynamic Change and capture management Information?*, 4th International Conference on Cooperative Information Systems, CoopIS'99, Edinburgh, Scotland, 1999.

[Alagic97] Alagic, S.: *ODMG Object Model: Does it Make Sense?* Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'97, Atlanta, Georgia, USA, Oct 1997

[Abate02] Abate, A. F., Esposito, A., Grieco, N., Nota, G.: *Workflow Performance Evaluation through WPQL*, 14th international conference on Software engineering and knowledge engineering, Ischia, Italy, Jul 2002.

[Atkinson87] Atkinson, M.P., Buneman, P.: *Types and Persistence in Database Programming Languages*, ACM Computing Surveys 19(2):105-190, ACM (1987)

[Bliziuk05] Blizniuk, G., Momotko, M., Nowicki, B., Strychowski, J.: *The EWD-P System Polish Government – European Commission Interoperability Achieved*, 38th Hawaii International Conference on Computer Science, HICSS'38, Honolulu, Hawaii, Jan 2005.

[BPEL4WS03] IBM developer works: *Business Process Execution Language for Web Services*, ver. 1.1, May 2003.

[BPML02] Business Process Management Initiative: *Business Process Modelling Language*, Nov 2002.

[BPMN04] Business Process Management Initiative: *Business Process Modelling Notation*, ver. 1.0, May 2004.

[Buttler02] Buttler Group, *Business process Management report – on the basis Cap Gemini* Ernst & Young survey, 2002.
http://www.butlergroup.com/reports/bpm/mansum.asp

[Cardoso02] Cardoso, J., Miller, J., Sheth, A.: *Workflow Quality of Service*, the International Conference on Enterprise Integration and Modelling Technology and International Enterprise Modelling Conference (ICEIMT/IEMC'02), April 2002.

[Casati96] Casati, F., Ceri, S., Pernici, B., Pozzi, G.: *Deriving Active Rules for Workflow En-actmen*t; International Conference on Database and Expert System Applications, DEXA'1996, Zürich, Switzerland, 1996.

[Celko97] Celko, J. *SQL Puzzles and Answers*. Elsevier, ISBN 1558604537, 1997

[Ceri03] Ceri, S., Manolescu I.: *Constructing and integrating Data-Centric Web Applications; Methods, Tools and Techniques*, 29th International Conference on Very large Data Bases, VLDB'2003, Berlin, Germany, 2003.

[Ceri89] Ceri, S., Gottlob, G., Tanca, L. *What You Always Wanted to Know About Datalog (And Never Dared to Ask)*. IEEE Transactions on Knowledge and Data Engineering 1(1): 146-167, IEEE Computer Society (1989)

[Clark99a] Clark, J.: *XSL Transformations (XSLT)*, ver. 1.0, Nov 1999. http://www.w3.org/TR/1999/REC-xslt-19991116

[Clark99b] Clark J., DeRose S.: *XML Path Language (XPath)*, Nov 1999. http://www.w3.org/TR/1999/REC-xpath-19991116

[Date98] Date, C. J.: *Encapsulation Is a Red Herring. Intelligent Enterprise's Database on line, Programming and Design*, http://www.dbpd.com/vault/9809date.html, 1998

[Deutch98] Deutsch A., Fernandez M., Florescu D., Levy A., Suciu D.: *XML-QL: A Query Language for XML*, http://www.w3.org/TR/1998/NOTE-xml-ql-19980819, Aug 1998.

[Eder01] Eder, J., Paganos, E.: *Managing Time in Workflow Systems*, Workflow Handbook 2001, Layna Fischer (Ed.), Future Strategies Inc., Book Division, 2001, USA.

[Eder97] Eder, J.; Pozewaunig, H., Liebhart, W.: *ePERT: Extending PERT for Workflow Management Systems*, 1st East-European Conference on Advances in Databases and Information Systems (ADBIS'97), 1997.

[Eder99] Eder, J.; Panagos, E., Pozewaunig, H., Rabinovich, M.: *Time Management in Workflow Systems*, 3rd International Conference on Business Information System (BIS'99), p. 265-280, 1999.

[Eshuis01] R. Eshuis and R. Wieringa: *A real-time execution semantics for UML activity diagrams*, Fundamental Aspects of Software Engineering, FASE'01, April 2001.

[Fuhr00] Fuhr, N.: *XIRQL An Extension of XQL for Information Retrieval*, SIGIR 2000, http://www.haifa.il.ibm.com/sigir00-xml/final-papers/xirql.html.

[Hedge02] Hedge, A.: *Business Process Management Update*, Vol. 1, Issue 1, Oct. 2002.

[ICONS02] Rodan: *Intelligent Content Management System*, IST-2001-32429, 5th EC Framework Programme, www.icons.rodan.pl.

[ICONS04] Rodan: *ICONS project results*, IST-2001-32429, 5th EC Framework Programme, April, 2004.

[ISO14977] ISO/IEC 14977:1996, Information technology: *Syntactic metalanguage, Extended BNF*, Stage 90.92, 2001.

[jFlow98] OMG BODTF RFP#2 Submission: *Workflow Management Facility*, OMG bom/98-06-07, 1998.

[Junginger04] Junginger, S., Kuhn, H., Bayer, F., Karagiannis, D.: *Workflow-based Business Monitoring*, published in Workflow handbook 2004, Future Strategies Inc., 2004.

[List03] List B., Schiefer J., Bruckner R.: *Process Data Store: A Real-Time Data Store for Monitoring Business Processes*, 14th International Conference on Database and Expert Systems DEXA'2003, , Prague, Czech Republic, Sept 2003.

[Lotus01] Lotus Development Corporation: *Lotus Workflow Process Designer's Guide*, ver. 3.0, 2001.

[Melton01] Melton, J., Simon, A., R., Gray, J.: *SQL:1999 - Understanding Relational Language Components*, Morgan Kaufmann Publishers, 2001.

[Melton93] Melton, J., Simon, A.: *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, 1993.

[Momotko00] Momotko, M.: *Pracuj zgodnie z procedurami – implementacja procesów pracy w systemie OfficeObjects®DocMan* (in Polish), 2nd Polish Conference on Software Engineering, KKIO'2000, Zakopane, Poland, Oct 2000.

[Momotko01] Momotko, M.: *OfficeObjects®WorkFlow – komponent wspierający zarządzanie procesami pracy* (in Polish), 3rd Polish Conference on Software Engineering, KKIO'2001, Otwock, Poland, Oct 2001.

[Momotko02a] Momotko, M., Subieta, K., *Dynamic change of Workflow Participant Assignment*. 6th East-European Conference on Advances in Database Information Systems, ADBIS'2002, Bratislava, Slovakia, 2002.

[Momotko02b] Momotko, M., Nowicki, B.: *Testowanie wbudowane – Europejski eksperyment walidacyjny* (in Polish), 4th Polish Conference on Software Engineering, KKIO'2002, Poznań, Poland, Oct 2002.

[Momotko03a] Momotko, M., Zalewska, L.: *Component+ Butli-in Testing – a Technology for Testing Software Components*, 5th Polish Conference on Software Engineering, KKIO'2003, Szklarska Poręba, Poland, Oct 2003.

[Momotko03b] Momotko, M., Nowicki, B.: *Visualisation of (Distributed) Process Execution based on Extended BPMN*, 14th International Conference on Database and Expert Systems, DEXA'2003, Web Based Collaboration workshop, WBC'2003, Prague, Czech Republic, Sept 2003.

[Momotko04] Momotko, M., Subieta, K.: *Business process Query Language – a Way to Make Workflow Processes More Flexible*, 8th East-European Conference on Advances in Database Information Systems, ADBIS'2004, Budapest, Hungary, 2004.

[Moore00] Moore, J., Inder, R., Chung, P., Macintosh, A., Stader, J., *Who Does What? Matching Agents to Tasks in Adaptive Workflow*, 2nd International Conference on Enterprise Information Systems, ICEIS'2000, Stafford, July 2000.

[ODMG00] Object Data Management Group: *The Object Database Standard ODMG, Release 3.0*. R.G.G.Cattel, D.K.Barry, Ed., Morgan Kaufmann, 2000.

[Plodzien00] J.Płodzień. *Optimization Methods in Object Query Languages*. Ph.D. Thesis. Institute of Computer Science, Polish Academy of Sciences , 2000, http://www.ipipan.waw.pl/~jpl

[Robbie00] Robie J., Chamberlin D., Florescu D.: *QUILT: an XML query language* XML Europe, Jun 2000.
http://www.gca.org/papers/xmleurope2000/papers/s08-01.html

[Robbie98] Robie J., Lapp J., Schach D.: *XML Query Language*, http://www.w3.org/TandS/QL/QL98/pp/xql.html, Sep 1998.

[Rumbaugh99] Rumbaugh J., Jacobson I., Booch G.: *The Unified Modeling Language Reference Manual*, Addision Wesley, 1999.

[Sadiq00] Sadiq, S.: *Handling Dynamic Schema Changes in Workflow Processes*, 11th Australian database Conference, Canberra, Australia, 2000.

[Shapiro02] Shapiro, R.: *A Comparison of XPDL, BPML and BPEL4WS*, rough draft version 1.4, Cape Vision, 2002.

[Staffware99] Staffware: *Procedure Definer's Guide*; Staffware Plc, Issue 2, March 1999.

[Subieta04] Subieta, K.: *Theory and Construction of Object-Oriented Query Languages* Editors of the Polish-Japanese Institute of Information Technology, 2004.

[Subieta95a] Subieta K., Beeri, C., Matthes, F., Schmidt, J.,W., *A Stack-Based Approach to Query Languages*, East-West Database Workshop, 1994, Springer Workshops in Computing, 1995.

[Subieta95b] Subieta K., Kambayashi Y., and Leszczyłowski J.: *Procedures in Object-Oriented Query Languages*, 21th International Conference on Very large Data Bases, VLDB'1995, Zurich, Switzerland, 1995.

[Subieta97] Subieta, K.: *Object-Oriented Standards: Can ODMG OQL be Extended to a Programming Language?* Cooperative Databases and Applications, World Scientific, 459-468, 1997.

[Swenson98] Swenson, K., Netscape Communications Corp., *Simple WorkFlow Access Protocol*, Aug 1998.

[Vector00] Vector: *Vector Concepts Survey of 200 North American Mid-sized Manufacturers Reveals Workflow, E-Commerce, Multi-currency Challenges*, Vector, 2000. http://www.bluecatdesign.com/w4/vector/new.html

[Weske99] Weske, M., Vossen, G.: *Flexibility and Cooperation in Workflow Management Systems*, Handbook on Architectures of Information Systems., pp 359–379. Berlin, Springer, 1998.

[WfMC-TC-1003] Workflow Management Coalition: *Workflow Reference Model*; WfMC-TC-1003, version 1.1, Jan 1995.

[WfMC-TC-1009] Workflow Management Coalition: *Workflow Management Application – Programming Interface (Interface 2 & 3) Specification*, ver. 2.0; Jul 1998.

[WfMC-TC-1011] Workflow Management Coalition: *Terminology & Glossary*, WfMC-TC-1011, version 3.0, Feb 1999.

[WfMC-TC-1012] Workflow Management Coalition: *Workflow Standard – Interoperability, Abstract specification*, WfMC-TC-1012, version 2.0b, draft. Nov 1999.

[WfMC-TC-1015] Workflow Management Coalition: *Audit Data Specification*, WfMC-TC-1015, version 1.1, Sep 1998.

[WfMC-TC-1016-P] Workflow Management Coalition: *Interface 1: Process Definition Interchange, Process Model*, WfMC-TC-1016-P, version 1.1, Oct 1999.

[WfMC-TC-1023] Workflow Management Coalition: *Wf-XML Binding*, WfMC-TC-1023, version 1.1, Nov 2001.

[WfMC-TC-1025] Workflow Management Coalition: *Workflow Process Definition Interface - XML Process Definition Language*, WfMC-TC-1025, version 1.0, Oct 2002.

[Winslet02] Winslett, M. *Interview with David Maier*. SIGMOD Record 31(4): 76-85 (2002) http://www.acm.org/sigmod/record/issues/0212/maier-oct-16.pdf

[WSCI02] BEA Systems, Intalio, SAP, Sun Microsystems, *Web Service Choreography Interface*, version 1.0, 2002. http://ifr.sap.com/wsci/specification/wsci-spec-10.html

[WSDL01] W3C Consortium: *Web Service Description Language*, version 1.1, W3C, March 2001.

[XQuery03] World Wide Web Consortium (W3C): *XQuery 1.0: An XML Query Language*, working draft , Nov 2003.

Appendix A: BPQL Syntax in EBNF

The formal grammar of the BPQL syntax is given in EBNF13.

Rule			Additional assumptions
<query>	::=	<literal>	
	|	<name>	<name> ∈ N
	|	<algExpression>	Algebraic expression
	|	<logExpression>	logical expression
	|	(<queryList>)	Bag
	|	<query> <collectionOp> <query>	collection operators
	|	**struct**(<queryList>)	Structure
	|	**bag**(<queryList>)	Bag
	|	**sequence**(<queryList>)	sequence
	|	<query> **as** <aliasName>	aliasing, <aliasName> ∈ N
	|	<query> **group as** <aliasName>	aliasing, <aliasName> ∈ N
	|	<query> **.** <query>	selection
	|	<query> **where** <query>	projection
	|	<query> **join** <query>	dependent join
	|	**for some** <query> **(** <query> **)**	existential quantifier
	|	**for all** <query> **(** <query> **)**	universal quantifier
	|	<query> **order by** <query>	sorting operator
	|	<procedureCall>	procedure call
<algExpression>	::=	<algSum>	algebraic expression
<algSum>	::=	<algProduct> { [**+** | **-**] <algSum>}	
<algProduct>	::=	<algSingleExpression> { [* | / | %] <algProduct>}	
<algSingleExpression>	::=	**(** <query> **)**	
<logExpression>	::=	<logSum>	logical expression
<logSum>	::=	<logProduct> { **or** <logSum>}	
<logProduct>	::=	<logSingleExpression> { **and** <logProduct> }	
<logSingleExpression>	::=	**not** <logExpression>	
	|	**(** <query> **)**	
	|	<logCondition>	
<logCondition>	::=	<leftSide> <opComp> <rightSide>	
<opComp>	::=	**<** | **<=** | **=** | **=>** | **>** | **<>** | **like**	
<leftSide>	::=	<algExpression>	
<rightSide>	::=	<algExpression>	
<collectionOp>	::=	**union**	set operators
	|	**intersect**	
	|	**diff**	
	|	**in**	
<collection>	::=	**struct**(<queryList>)	Type of sets
	|	**bag**(<queryList>)	
	|	**sequence**(<queryList>)	
<procedureCall>	::=	<pName>	<pName> ∈ N
	|	<pName> **( )**	
	|	<pName> **(**<queryList>**)**	
<procedure>	::=	<pName> <statementBlock>	
	|	<pName> **( )** <statementBlock>	
	|	<pName> **(** pParameterList>**)** <statementBlock>	
<pParameterList>	::=	<pParameter> {, <pParameter>}	

[13] Names written in boldface represent terminals.

<pParameter>	::=	<parName>	<parName> ∈ N
	|	**out** <parName>	
<statementBlock>	::=	<statement>**;**	
<statementBlock>	::=	**{** <statement >**;** { <statement> **;**} **}**	
<statement>	::=	<create>	
	|	<assignment>	
	|	<forEach>	
	|	<ifThenElse>	
	|	<switchCase>	
	|	<whileDo>	
	|	<doWhile>	
	|	<procedureCall>	
	|	**return** {<query>}	
	|	**break**	
	|	**continue**	
<create>	::=	**create** <query>	
<assignment>	::=	<lValue> **:=**<rValue>	
<lValue>	::=	<query>	<query> returns a reference to a single object
<rValue>	::=	<query>	
<forEach>	::=	**for each** <query> **do** <statementBlock>	
<ifThenElse>	::=	**if** <query> **then** <statementBlock>	<query> returns a Boolean object
<switchCase>	::=	**switch (**<query>**) {** <caseList> **}**	<query> returns a Boolean object
<caseList>	::=	<case> {<case>} {**case default**: <statementBlock>}	
<case>	::=	**case** <literal> **:** <statementBlock>	
<whileDo>	::=	**while (**<query>**)** <statementBlock>	<query> returns a Boolean object
<doWhile>	::=	**do** <statementBlock> **while (**<query>**)**	<query> returns a Boolean object
<symbol>	::=	<objName> {. <objName>}	// <objName> ∈ N
<literal>	::=	<text>	An alpha-numeric string surrounded by apostrophes
	|	<integer>	
	|	<float>	
	|	<boolean>	
<boolean>	::=	**true**	
	|	**false**	

Appendix B: Definition of the Built-in Workflow Monitoring Functions

This appendix includes the BPQL definition of the built-in workflow monitoring functions specified in chapter 5 . The code of every function is supported with the header written in the JavaDoc style and including specification of the function parameters and its result.

1. Start Activities

```
/** Determines start activities for given process definitions.
 * @par parProcessDef a bag of the processed process definitions.
 * @return a bag of process definitions (resProcessDef attribute) and their start activites (resStartActivity
attribute).
 */
procedure StartActivity(parProcessDef) {
   return (parProcessDef as e).
        (
                (e as resProcessDef),
                ((e.ConsistsOf.Activity where (count(From)=0)) as resStartActivity)
        );
}
```

2. End Activities

```
/** Determines end activities for given process definitions.
 * @par parProcessDef a bag of the processed process definitions.
 * @return a bag of process definitions (resProcessDef attribute) and their end activites (resEndActivity
attribute).
*/
procedure EndActivity(parProcessDef) {
   return (parProcessDef as e).
        (
                (e as resProcessDef),
                ((e.ConsistsOf.Activity where (count(To)=0)) as resEndActivity)
        );
}
```

3. Successor Activities

```
/** Determines the successors of given activities.
 * @par parActivities a bag of the processed activities.
 * @par parKind the type of successors to be retrieved.
*          If 'all' is chosen, the function returns all (i.e. direct and indirect) successors.
 *          If 'direct' is chosen, the function returns only direct successors.
 * @return a bag of the determined successors (activities).
 */
procedure SuccActivity(parActivities, parKind) {
  // since this procedure is recursive, first check the stop condition
  if (count(parActivities) = 0) then
          return bag(); // an empty collection
// process data according to the kind of required successors
  switch (parKind) {
     case 'all': { // return direct and indirect successors
           return distinct((parActivities.To.Transition.To.Activity as p).
                                    (p union SuccActivity( p, parKind)));
      }
      case 'direct': { return distinct(parActivities.To.Transition.To.Activity); } // only direct successors
      case default: { return bag(); }  // otherwise return only an empty bag
  }
}
```

4. Predecessor Activities

```
/** Determines the predecessors of given activities.
 * @par parActivities a bag of the processed activities.
 * @par parKind the kind of predecessors to be retrieved.
*          If 'all' is chosen, the function returns all (i.e. direct and indirect) predecessors.
 *          If 'direct' kind is chosen, the function returns only direct predecessors.
* @return a bag of the determined predecesors (activities).
*/
procedure PredActivity (parActivities, parKind) {
  // since this procedure is recursive, first check the stop condition
  if (count(parActivities) = 0) then return bag(); // an empty collection
// process data according to the kind of required predecessors
  switch (parKind) {
     case 'all': {  // return direct and indirect predecessors
              return distinct((parActivities.From.Transition.From.Activity as p).
                                          (p union PredActivity( p.From.Transition.From.Activity));
      }
      case 'direct': { return distinct(parActivities.From.Transition.From.Activity); } // only direct predecessors
      case default: {  return bag(); }
  }
}
```

5. Start Activity Instance

```
/** Determines the start activity instances for given process instances.
 * @par parProcessInst a bag of the processed process instances.
* @return a bag of process instances (resProcessInst attribute ) together with
* their start activity instances (resStartActivityInst attribute).
*/
procedure StartActivityInst(parProcessInst) {
   return (parProcessInst as e).
            (
                (e as resProcessInst),
               ((e.ConsistsOf.ActivityInst  where (InstanceOf.Activity.(count(To) = 0))) as resStartActivityInst)
            );
}
```

6. End Activity Instance

```
/** Determines the end activity instances for given process instances.
 * @par parProcessInst a bag of the processed process instances.
* @return a bag of process instances (resProcessInst attribute) together with
* their end activity instances (resEndActivityInst attribute).
*/
procedure EndActivityInst(parProcessInst) {
  return (parProcessInst as e).
          (
             (e as resProcessInst),
             ((e.ConsistsOf.ActivityInst  where (InstanceOf.Activity.(count(From) = 0))) as resEndActivityInst)
          );
}
```

7. Successor Activity Instances

```
/** Determines the successors for given activity instances.
* @par parActivityInst a bag of the processed activity instances.
 * @par parKind the kind of successors to be retrieved.
*          If 'all' is chosen, the function returns all (i.e. direct and indirect) successors.
 *          If 'direct' is chosen, the function returns only direct successors.
 * @returns a bag of the determined successors (activity instaces)
*/
procedure SuccActivityInst(parActivityInst, parKind) {
 // since this procedure is recursive, first check the stop condition
 if (count(parActivityInst) = 0) then  return bag();
 // process data according to the kind of required successors
 switch (parKind) {
     case 'all': { // return direct and indirect successors
        return distinct((parActivityInst.To.TransitionInst.To.ActivityInst as a).
                                          (a union SuccActivityInst( a.To.TransitionInst.To.ActivityInst));
    }
    case 'direct': { return distinct(parActivityInst.To.TransitionInst.To.ActivityInst); } // only direct successors
    case default: { return bag();}
 }
}
```

8. Predecessor Activity Instances

```
/** Determines predecessors for given activity instances.
* @par parActivityInst a bag of the processed activity instances.
 * @par parKind the kind of predecessors to be retrieved.
*          If 'all' is chosen, the function returns all (i.e. direct and indirect) predecessors.
 *          If 'direct' is chsoen, the function returns only direct predecessors.
* @returns a bag of the determined predecessors (activity instances).
*/
procedure PredActivityInst(parActivityInst, parKind) {
  // since this procedure is recursive, first check the stop condition
  if (count(parActivityInst) = 0) then return bag();
  // process data according to the kind of required predecessors
  switch (parKind) {
     // all predecessors
     case 'all': {  // return direct and indirect predecessors
                  return distinct((parActivityInst.From.TransitionInst.From.ActivityInst as a).
                                          (a union PredActivityInst( a.From.TransitionInst.From.ActivityInst));
      }
      // only direct predecessors
      case 'direct': { return distinct(parActivityInst.From.TransitionInst.From.ActivityInst); }
      case default: { return bag(); }
  }
}
```

9. Current Amount

```
/**Determines the current amount of tasks (activity instances) assigned to given resources (workflow
participants).
* @par parResource a bag of the processed resources
 * @return a bag of resources (resResources attribute) together with
 * the currrent amount of their tasks (resCount attribute).
*/
procedure WorkloadCount(parResource) {
  return (parResource as e).
            (
              (e as resResource),
              (count(e.IsPerformer.Performer.Performs.ActivityInst
                        where (currState.opSuperState) in('Created','Running')
               ) as resCount)
            );
}
```

10. Expected Processing Time

```
/** Calculates the total time needed to complete tasks assigned to given resources (workflow participants).
* @par parResource a bag of the processed resources
* @return a bag of resources (resResources attribute) together with
 * the currrent amount of their tasks (resExpectedTime attribute).
*/
procedure WorkloadExpectedTime( parResource) {
  return (parResource as e).
            (
              ( e as resResource),
               (sum((e.IsPerformer.Performer.Performs.ActivityInst
                        where (currState.opSuperState) in('Created','Running')).
                                 (InstanceOf.Activity.duration - duration)
                 ) as resExpectedTime)
            );
}
```

11. Overall Expected Cost

```
/** Calculates the overall expected cost to complete the tasks assigned to given resources (workflow participants).
* @par parResource a bag of the processed resources
* @return a bag of resources (resResources attribute) together with
 * the overall expected cost of their tasks (resExpectedCost attribute).
*/
procedure WorkloadExpectedCost(parResource)  {
return (parResource as e).
            (
              (e as resResource),
              (sum((e.IsPerformer.Performer.Performs.ActivityInst
                        where (currState.opSuperState) in('Created','Running')).
                                 (InstanceOf.Activity.cost - cost)
                 ) as resExpectedCost)
            );
}
```

12. Accumulated Priority

```
/** Calculates the accumulated priority of the tasks assigned to given resources (workflow participants).
* @par parResource a bag of the processed resources
* @return a bag of resources (resResources attribute) together with
 * the calculated priority of their tasks (resAccPriority attribute).
*/
procedure WorkloadAccPriority(parResource)  {
  return (parResource as e).
            (
              (e as resResource),
              (sum((e.IsPerformer.Performer.Performs.ActivityInst
                        where (currState.opSuperState) in('Created','Running')).priority
               ) as resAccPriority)
            );
}
```

13. Smallest Current Amount

```
/** Selects workflow participants (resources) from a bag of given resources with the minimal current workload
in
*  terms of the number of the assigned tasks. If there is more than one such participant, all are returned.
* @param parResource a bag of the processed Resources
* @return a bag of the selected Resources.
*/
procedure BestPerformerCount(parResource) {
   return  (WorkloadCount(parResource) group as rc).
           (rc where resCount = min( rc. resCount)).resResource;
}
```

14. Shortest Total Processing Time

```
/** Selects workflow participants (resources) from a bag of given resources with minimal current workload in terms
* of the expected total time to complete the assigned tasks. If there is more than one participant, all are returned.
* @param parResource a bag of the processed Resources
* @return a bag of the selected Resources.
*/
procedure BestPerformerExpectedTime(parResource) {
  return   (WorkloadExpectedTime(parResource) group as et).
           (et where resExpectedTime  =  min(et.resExpectedTime)).resResource;
}
```

15. Minimal Overall Cost

```
/** Selects workflow participants (resources) from a bag of given resources with minimal current workload
*  in terms of the minimal expected overall cost to complete the assigned tasks.
* If there is more than one such workflow participant, all are returned.
* @param parResource a bag of the processed Resources
* @return a bag of the selected Resources.
*/
procedure BestPerformerExpectedCost(parResource) {
return    (WorkloadExpectedCost (parResource) group as ec).
          (ec where resExpectedCost  =  min(ec.resExpectedCost)).resResource;
}
```

16. Smallest Accumulated Priority

```
/** Selects workflow participants (resources) from a bag of given resources with minimal current workload
*  in terms of the smallest accumulated priority to complete the assigned tasks.
* If there is more than one such workflow participant, all are returned.
* @param parResource a bag of the processed Resources
* @return a bag of the selected Resources.
*/
procedure BestPerformerAccPriority(parResource) {
return (WorkloadAccPriority(parResource) group as ap).
          (ap where resAccPriority  =  min(ap.resAccPriority)).resResource;
}
```

17. Activity Instance - Duration of a State and Superstate

```
/** Calculates a sum of periods of time for activity instances and a set of operational states/superstates.
 * @par parActivityInst a bag of activity instances.
 * @par parOpSuperState a bag of the superstate names
 * @par parOpState a bag of the state names
 * @return a bag of activity instances (resActivityInst attribute) togeteher with
*  the calculated period which they spent in a given state or superstate (resDuration attribute).
 */
procedure ActivityInstStateDuration( parActivityInst, parOpSuperState, parOpState) {
  return (parActivityInst as e).
          (
           ( e as resActivityInst),
           (sum(
                   (e.ProcessedIn.ActivityInstState
                    where
                           (count(parOpSuperState) = 0 or opSuperState in parOpSuperState) and
                           (count(parOpState) = 0 or opState in parOpState)
                   ).
                   ( if (finishDate > 0) then (finishDate – startDate);    // the state has already been completed
                            else (CurrDate() – startDate);                 // this is the current state
                   )
           ) as resDuration)
          );
}
```

18. Activity Instance – Expected Processing Time

```
/** Calculates the expected processing time needed to complete given tasks (activity instances).
* @par parActivityInst a bag of the processed activity instances
* @return a bag of activity instances (resActivityInst attribute) together with
 * the expected processing time (resExpectedTime attribute).
*/
procedure ActivityInstProcessingTime(parActivityInst) {
  return (parActivityInst as e).
          (
           ( e as resActivityInst),
           ((e.InstanceOf.Activity.duration - ActivityInstStateDuration( e, bag( ), bag( )).resDuration
            ) as resExpectedTime)
          );
}
```

19. Activity Instance - Waiting Time

```
/** Calculates the waiting time for given tasks (activity instances).
* @par parActivityInst a bag of the processed activity instances
* @return a bag of activity instances (resActivityInst attribute) together with
 * the calculated waiting time (resWaitingTime attribute). */
procedure ActivityInstWaitingTime( parActivityInst) {
  return (parActivityInst as e).
          (
            (e as resActivityInst),
            (ActivityInstStateDuration( e, 'Created', bag( )).resDuration as resWaitingTime)
          );
}
```

20. Activity Instance – Working time

```
/** Calculates the working time for given tasks (activity instances).
* @par parActivityInst a bag of the processed activity instances
* @return a bag of activity instances (resActivityInst attribute) together with
 * the calculated working time (resWorkingTime attribute).
*/
procedure ActivityInstWorkingTime( parActivityInst) {
  return (parActivityInst as e).
          (
            (e as resActivityInst),
            (ActivityInstStateDuration( e, 'Running', bag( )).resDuration as resWorkingTime)
          );
}
```

21. Process Instance – Waiting time

```
/** Calculates the waiting time for given process instances.
* @par parProcessInst a bag of the processed process instances
* @return a bag of process instances (resProcessInst attribute) together with
 * the calculated waiting time (resWaitingTime attribute). */
procedure ProcessInstWaitingTime( parProcessInst) {
  return (parProcessInst as e).
          (
            (e as resProcessInst),
            (sum(ActivityInstWaitingTime(e.ConsistsOf.ActivityInst). resWaitingTime) as resWaitingTime)
          );
}
```

22. Process Instance – Working time

```
/** Calculates the working time for given process instances.
* @par parActivityInst a bag of the processed process instances
* @return a bag of process instances (resProcessInst attribute) together with
 * the calculated working time (resWorkingTime attribute).
*/
procedure ProcessInstWorkingTime( parProcessInst) {
  return (parProcessInst as e).
          (
            (e as resProcessInst),
            (sum(ActivityInstWorkingTime(e.ConsistsOf.ActivityInst). resWorkingTime) as resWorkingTime)
          );
}
```

23. Activity – Waiting Time

```
/** Calculates the average value of the total waiting time for given activities.
 * @par parActivity a bag of given actvities.
 * @return a bag of activities (resActivity attribute) together with
 * the average value of the total waiting time for these activities (resWaitingTime attribute).
 */
procedure ActivityWaitingTime( parActivity) {
  return (parActivity as e).
          (
            (e as resActivity),
            (avg(ActivityInstWaitingTime(
                    e.InstantiatedAs.ActivityInst                    // all instantiation of a given activity
                    where ((currState.opSuperState) = 'Finished')    // which have been already finished
              ).resWaitingTime) as resWaitingTime)
          );
}
```

24. Activity – Working Time

```
/** Calculates the average value of the total working time for given activities.
 * @par parActivity a bag of given actvities.
 * @return a bag of activities (resActivity attribute) together with
 * the average value of the total working time for these activities (resWorkingTime attribute).
 */
procedure ActivityWorkingTime( parActivity) {
  return (parActivity as e).
        (
          (e as resActivity),
          (avg(ActivityInstWorkingTime(
                e.InstantiatedAs.ActivityInst                    // all instantiation of a given activity
                where ((currState.opSuperState) = 'Finished')    // which have been already finished
           ).resWorkingTime) as resWorkingTime)
        );
}
```

25. Process – Waiting Time

```
/** Calculates the average value of the total waiting time for given process definitions.
 * @par parProcessDef a bag of given process definitions.
 * @return a bag of process definitions (resProcessDef attribute) together with
 * the average value of the total waiting time for these processes (resWaitingTime attribute).
 */
procedure ProcessWaitingTime(parProcessDef) {
  return (parProcessDef as e).
        (
          (e as resProcessDef),
          (avg(ProcessInstWaitingTime(
                e.InstantiatedAs.ProcessInst                     // all instantiation of a given
process
                where ((currState.opSuperState) = 'Finished')    // which have been already
finished
           ).resWaitingTime) as resWaitingTime)
        );
}
```

26. Process – Working Time

```
/** Calculates the average value of the total working time for given process definitions.
 * @par parProcessDef a bag of given process definitions.
 * @return a bag of process definitions (resProcessDef attribute) together with
 * the average value of the total working time for these processes (resWorkingTime attribute).
 */
procedure ProcessWorkingTime(parProcessDef) {
  return (parProcessDef as e).
        (
          (e as resProcessDef),
          (avg(ProcessInstWorkingTime(
                e.InstantiatedAs.ProcessInst                    // all instantiation of a given
process
                where ((currState.opSuperState) = 'Finished')   // which have been already
finished
           ).resWorkingTime) as resWorkingTime)
        );
}
```

27. Activity Instance Cost

```
/** Calculates the cost for given activity instances.
 * @par parActivityInst a bag of activity instances.
 * @return a bag of activity instances (resActivityInst attribute) together with
 * their cost (resCost attribute).
*/
procedure ActivityInstCost( parActivityInst) {
  return (parActivityInst as e).
        (
           (e as resActivityInst),
          // calculate as the product of working time (converted from milliseconds to hours)
          // and the resource-hour rate
           (ActivityInstWorkingTime(e).
                (resActivityInst.PerformedBy.Performer.Represents.Resource.hourRate *
                  resWorkingTime / (1000*60*60))
           as resCost)
        );
}
```

28. Process Instance Cost

```
/** Calculates the cost for given process instances.
 * @par parProcessInst a bag of process instances.
 * @return a bag of process instances (resProcessInst attribute) together with
 * their cost (resCost attribute).
*/
procedure ProcessInstCost( parProcessInst) {
  return (parProcessInst as e).
          (
            (e as resProcessInst),
            // calculate as the sum of the cost for all activity instances.
            (sum(ActivityInstCost(e.ConsistsOf.ActivityInst).resCost)
             as resCost)
          );
}
```

29. Activity Cost

```
/** Calculates the average cost for given activities.
 * @par parActivity a bag of activities.
 * @return a bag of activities (resActivity attribute) together with
 * their cost (resCost attribute).
*/
procedure ActivityCost( parActivity) {
  return ( parActivity as e).
          (
            (e as resActivity),
            // calculate as the average of the cost for all activity instances.
            (avg(ActivityInstCost(e.InstantiatedAs.ActivityInst).resCost) as resCost)
          );
}
```

30. Process Cost

```
/** Calculates the cost for given process definitions.
 * @par parProcessInst a bag of process definitions.
 * @return a bag of processes (resProcessDef attribute) together with
 * their cost (resCost attribute).
*/
procedure ProcessCost(parProcessDef) {
  return (parProcessDef as e).
          (
            (e as resProcessDef),
            // calculate as the average of the cost for all process instances.
            (avg(ProcessInstCost(e.InstantiatedAs.ProcessInst).resCost) as resCost)
          );
}
```

31. Activity Reliability

```
/** Calculates the reliability for given activities.
 * @par parActivity a bag of given activities.
 * @return a bag of activities (resActivity attribute) together with
 * their reliability parameter (resReliability attribute).
* Reliability is represented as a float object with value between 0.. 1.
*/
procedure ActivityReliability( parActivity) {
  return (parActivity as e).
          (
            (e as resActivity),
            // calculate as 1 – count(activity inst terminated)/ count(act inst completed)
            (e.(
                  1 –
                  count(InstantiatedAs.ActivityInst where ((currState.opState) = 'Terminated'))/
                  count(InstantiatedAs.ActivityInst where ((currState.opState) = 'Completed'))
             ) as resReliability)
          );
}
```

32. Process Reliability

```
/** Calculates the relaibility for given process definitions.
 * @par parActivity a bag of given process definitions.
 * @return a bag of processes (resProcessDef attribute) together with
 * their reliability parameter (resReliability attribute).
* Reliability is represented as a float object with value between 0.. 1.
*/
procedure ProcessReliability( parProcessDef) {
  return (parProcessDef as e).
          (
            (e as resProcessDef),
           // calculate as 1 – count(activity inst terminated)/ count(act inst completed)
           (e.(
                    1 –
                    count(InstantiatedAs.ProcessInst where ((currState.opState) = 'Terminated'))/
                    count(InstantiatedAs.ProcessInst where ((currState.opState) = 'Completed'))
            ) as resReliability)
          );
}
```

Appendix C: BPQL Manager Interfaces

This appendix specifies interfaces and classes provided by BPQL Manager. Less important setter and getter methods have been omitted.

Interface BPQLMetaDataManager

This interface specifies functions to retrieve information about the operators, functions and other BPQL constructs. These functions are inherited from the *BPQLFunctionDefList* interface. It also provides functions to add and remove a user-defined built-in BPQL function.

Inherits: *BPQLFunctionDefList* interface

Methods	**public String addFunction(**BPQLFunctionDef functionDef**) throws** BPQLException; Add a new user-defined built-in BPQL function. **Parameters** *functionDef* – the definition of the added function represented as a BPQLFunctionDef object. **Returns** The identifier of the added function. **Throws** BPQLException if (i) a given function already exists in the BPQL repository, (ii) there is at least one BPQL query within a process definition which depends on the function. **public void removeFunction(**String functionId**) throws** BPQLException; Remove a user-defined built-in BPQL function. **Parameters** *functionId* – the identifier of the function to be removed. **Throws** BPQLException if (i) a given function does not exist in the BPQL repository. (ii) there is at least one BPQL query within a process definition which depends on the function.

Interface BPQLFunctionDefList

This interface specifies functions to retrieve information about the operators, functions and other BPQL constructs.

Methods	**public void open(**String where, String order**) throws** BPQLException; Set the criteria to retrieve and sort the operators, functions and other BPQL constructs. **Parameters** *where* – a BPQL query to select the mentioned BPQL elements. It may include the

	function attributes, Boolean operators and comparison operators. *order* – a list of function attributes separated by comma. Their order determines the order of the returned objects. **Throws** BPQLException if (i) a given *where* expression has an error, (ii) a given *sort* expression has an error. **public Collection get(**int fromIndex, int toIndex**) throws** BPQLException; Get a collection of the operators, functions and other BPQL constructs. The function retrieves the objects starting from fromIndex up to toIndex. Other objects will not be returned. If there is less than (toIndex-fromIndex+1) objects, only those available are returned. **Parameters** *fromIndex* – the position which the objects will be extracted from. *toIndex* – the position which the objects will be extracted to. **Returns** A collection of BPQLFunction objects. **Throws** BPQLException if it is not possible to create the collection. **public long getCount() throws** BPQLException; Get the number of the operators, functions and other BPQL constructs which satisfy the query defined in the *open* method. **Returns** The number of retrieved objects. **Throws** BPQLException if (i) it is not possible to get the number of objects. **public void close()** Clear the *where* and *sort* criteria set in the open method.

Class BPQLFunctionDef

This class represents the definition of a BPQL function. Some attributes are inherited from BPQLBasicElement class.

Inherits: *BPQLBasicElement* class

Attributes	**private int** argumentCount**;** The number of the function parameters. **private Collection** argumentList**;** A list of the function parameters. **private** BPQLType resultType; The function result type.

Methods	**public int addParameter(**BPQLFunctionParam param**) throws** BPQLException; Add a new parameter to the BPQL function. **Parameters** *param* – the definition of the added function parameter represented as a BPQLFunctionParam object. **Returns** The identifier of the added function parameter. **Throws** BPQLException if (i) a given function parameter already exists in the BPQL repository, (ii) the object may not be added because of an internal error. **public void removeParamater(**String parameterId**) throws** BPQLException; Remove a function parameter. **Parameters** *parameterId* – the identifier of the function parameter to be removed. **Throws** BPQLException if (i) a given function parameter does not exist in the BPQL repository. (ii) there is at least one BPQL query within a process definition which depends on the function.

Class BPQLBasicElement

This class represents the definition of a core BPQL element.

Attributes	**private** String id**;** The identifier of the element. **private** String name**;** The element name. **private BPQLElementType type;** The type of the BPQL element. **private** String displayName;. The name of the element which will be displayed in a BPQL query when it will be defined.

Class BPQLFunctionParam

This class represents the definition of a BPQL function parameter.

Attributes	**private** String id**;** The identifier of the parameter. **private** String name**;** The parameter name. **private BPQLType type;** The parameter type.

Class BPQLEvaluator

This interface specifies functions to verify and evaluate a BPQL query.

Methods	**public boolean verify(**BPQLQuery query**) throws** BPQLException**;** Verify whether a given BPQL query is valid. **Parameters** *query* – a BPQL query which is verified. **Returns** true – query is valid, false – otherwise. **Throws** BPQLException if a given query has an error. **public Collection evaluate(**BPQLQuery query**) throws** BPQLException; Evaluate a given BPQL query. **Parameters** query – a BPQL query which is evaluated. **Returns** A collection of objects which satisfy the query. If there is no results an empty collection is returned. **Throws** BPQLException if a given query has e an error or objects can not be retrieved.

Class BPQLQuery

This class represents a BPQL query.

Attributes	**private** String value**;** The definition of the query..
Methods	**public** String getQuery() Get the query **public** void setQuery(String query) Set the query

Class BPQLQueryEval

This class represents evaluation of a BPQL query.

Attributes	**private** String processInstId; The identifier of the process instance on behalf of which the query is evaluated. Optional. It is used in ThisProcessInst function. **private** String activityInstId; The identifier of the activity instance on behalf of which the query is evaluated. Optional. It is used in ThisActivityInst function.

Class BPQLType

This class represents a BPQL type.

Attributes	**private** String name**;** The type name. **private** String javaType**;** The name of a Java programming language type which corresponds to this type.

Class BPQLElementType

This class represents the type of a BPQL element.

Attributes	**private** String name**;** The BPQL element name.

Class BPQL Exception

This class represents a BPQL exception generated during execution of a method specified in the interfaces and classed provided by the BPQL Manager.

Inherits: *Exception* class.

Appendix D: Metamodel Glossary

A

activity 11, 22
activity instance 12, 36
activity instance state 39
activity instance, end activity instance 106
activity instance, start activity instance 105
activity, atomic activity 11, 23
activity, compound activity 11, 24
activity, direct predecessor 105
activity, direct successor 104
activity, end activity 103
activity, predecessor 105
activity, routing activity 11, 25
activity, start activity 103
activity, successor 104
application 10, 11
application call 12, 43
attribute 44
attribute definition 43

C

container attribute 10, 36
container attribute definition 21

D

data 9
data object 12

E

event 11
event, external event 11
event, time event 11
event, workflow event 11

J

join 25
join, AND-Join 26
join, OR-Join 28
join, XOR-Join 27

O

object reference 41
object type 15
object type specification 29
object types 11

P

parallel routing 26
parameter 46
parameter definition 45
parameter, process definition 11
participant assignment 30
performer 42
process definition 10, 18
process enactment 32
process instance 12, 32
process instance state 12, 34

R

resource 9
resource definition 15
routing, alternative routing 27
routing, conditional routing 28

S

service 9
service declaration 15
split 25
split, AND-Split 26
split, OR-Split 28
split, XOR-Split 27
sub-process 12

T

transition 12, 29

transition condition 29
transition instance 12, 41

W

workflow control data 10
workflow expression, definition 46
workflow expression, evaluation 47
workflow participant 10, 12
workflow participant assignment 11

Appendix E: BPQL Glossary

A

aggregate functions 80
alias operator 82
arithmetic functions 79

B

bag 82
binder 74

C

collection and structure operators 81
collection functions 80
comparison operators 79
control flow operators 90

D

data storage 72
dependent join 86

E

environmental stack 72
environmental stack, bind 74
environmental stack, pop 74
environmental stack, push 74
existential quantifier 86

F

for each operator 89
function 92
function, context dependent 93

I

if-then-else 90
imperative construct 88

L

literal 69
loop operators 91

N

name 69
new object 89

P

procedure 92
projection 86

Q

query evaluation procedure 76
query evaluation procedure, algebraic operators 78
query evaluation procedure, literals and names . 77
query evaluation procedure, non-algebraic operators 84
query result stack 72
query result stack, empty 75
query result stack, pop 75
query result stack, push 75
query result stack, top 75

S

selection 85
sequence 82
sorting operator 88
statement 88
statement block 88
string functions 80
struct 82
switch-case 90

T

token 69
type coercions 80

U

universal quantifier 87
update, insert and delete operators 89

Appendix F: List of Figures

FIGURE 1.1 REQUIRED STEPS OF THE PROPOSED APPROACH 4
FIGURE 2.1 BASIC RELATIONS BETWEEN IT SYSTEM AND WFM SYSTEM 10
FIGURE 2.2 THE WORKFLOW PROCESS METAMODEL – CONCEPTUAL VIEW 14
FIGURE 2.3 MAPPING BETWEEN ELEMENTS OF IT SYSTEMS AND WFM SYSTEMS – CONCEPTUAL VIEW 16
FIGURE 2.4 SELECTED ELEMENTS OF THE IT SYSTEM - OBJECT DATABASE STRUCTURE VIEW 17
FIGURE 2.5 THE WORKFLOW METAMODEL FOR THE PROCESS DEFINITION – DATABASE SCHEMA VIEW 19
FIGURE 2.6 PARALLEL ROUTING EXAMPLE 26
FIGURE 2.7 ALTERNATIVE ROUTING EXAMPLE 27
FIGURE 2.8 THE CONDITIONAL ROUTING EXAMPLE 28
FIGURE 2.9 THE WORKFLOW METAMODEL FOR THE PROCESS ENACTMENT – DATABASE SCHEMA VIEW 33
FIGURE 2.10 PROCESS INSTANCE BEHAVIOURAL MODEL 34
FIGURE 2.11 ACTIVITY INSTANCE BEHAVIOURAL MODEL 39
FIGURE 3.1 AN EXAMPLE – A SIMPLIFIED METAMODEL 56
FIGURE 3.2 AN EXAMPLE - A SAMPLE OF DATA 57
FIGURE 3.3 AN XML REPRESENTATION OF THE EXAMPLE 58
FIGURE 3.4 A RELATIONAL REPRESENTATION OF A PART OF THE METAMODEL 62
FIGURE 4.1 ARCHITECTURE OF THE QUERY EVALUATION MECHANISM 75
FIGURE 4.2 THE ARITHMETIC STACK STATES FOR A SIMPLE EXPRESSION 77
FIGURE 4.3 QRES STACK - A SAMPLE OF A QUERY EVALUATION 77
FIGURE 4.4 QRES STACK AND QUERY EVALUATION 78
FIGURE 4.5 THE SYNTACTIC TREE FOR A SIMPLE QUERY 79
FIGURE 4.6 APPLICATION OF THE AS OPERATOR 85
FIGURE 4.7 APPLICATION OF THE GROUP AS OPERATOR 85
FIGURE 4.8 A SIMPLIFIED MODEL FOR ORDERING LAPTOPS 97
FIGURE 5.1 BPQL LAYERED ARCHITECTURE 100
FIGURE 5.2 TYPES OF THE BUILT-IN WORKFLOW MONITORING FUNCTIONS 101
FIGURE 5.3 THE PROCESS MODEL FOR ORDERING LAPTOPS 118
FIGURE 6.1 THE DESIGN TIME USE CASES FOR BPQL MANAGER 128
FIGURE 6.2 THE RUN TIME USE CASES FOR BPQL MANAGER 129
FIGURE 6.3 THE WFM + BPQL FUNCTIONAL ARCHITECTURE 131
FIGURE 6.4 THE DETAILED ARCHITECTURE OF THE BPQL MANAGER 133
FIGURE 6.5 THE MODEL OF THE PROVIDED INTERFACES 134
FIGURE 6.6 AN EXAMPLE OF THE EWD-P WORK LIST (IN POLISH) 138
FIGURE 6.7 ACTORS OF THE EWD-P ARCHITECTURE 139
FIGURE 6.8 A USER DIALOG TO SELECT BUITL-IN MONITORING FUNCTIONS (IN POLISH) 141
FIGURE 6.9 AN EXAMPLE OF HOW TO DEFINE WORKFLOW PARTICIPANTS WHO WILL BE INFORMED OF POSSIBLE TIME CONSTRAINTS VIOLATION (IN POLISH) 142

Appendix G: List of Tables

TABLE 2.1 SPECIFICATION OF THE RESOURCE ATTRIBUTES 17
TABLE 2.2 ASSOCIATIONS BETWEEN RESOURCE AND PERFORMER CLASSES 17
TABLE 2.3 SPECIFICATION OF THE (ADDITIONAL) USER ATTRIBUTES 17
TABLE 2.4 SPECIFICATION OF THE PROCESS DEFINITION ATTRIBUTES 20
TABLE 2.5 ASSOCIATIONS BETWEEN PROCESS DEFINITION AND OTHER WORKFLOW CLASSES 20
TABLE 2.6 SPECIFICATION OF THE PROCESS DEFINITION STATES 21
TABLE 2.7 SPECIFICATION OF THE (ADDITIONAL) CONTAINER ATTRIBUTE DEFINITION ATTRIBUTES 21
TABLE 2.8 ASSOCIATIONS BETWEEN CONTAINER ATTRIBUTE DEFINITION AND OTHER WORKFLOW CLASSES 22
TABLE 2.9 SPECIFICATION OF THE ACTIVITY ATTRIBUTES 22
TABLE 2.10 ASSOCIATIONS BETWEEN ACTIVITY AND OTHER WORKFLOW CLASSES 23
TABLE 2.11 ASSOCIATIONS BETWEEN ATOMIC ACTIVITY AND OTHER WORKFLOW CLASSES 23
TABLE 2.12 ASSOCIATIONS BETWEEN COMPOUND ACTIVITY AND OTHER WORKFLOW CLASSES 24
TABLE 2.13 SPECIFICATION OF THE PARAMETER MAPPING ATTRIBUTES 25
TABLE 2.14 ASSOCIATIONS BETWEEN PARAMETER MAPPING AND OTHER WORKFLOW CLASSES 25
TABLE 2.15 SPECIFICATION OF THE TRANSITION ATTRIBUTES 29
TABLE 2.16 ASSOCIATIONS BETWEEN TRANSITION AND OTHER WORKFLOW CLASSES 29
TABLE 2.17 SPECIFICATION OF THE OBJECT TYPE ATTRIBUTES 29
TABLE 2.18 ASSOCIATIONS BETWEEN OBJECT TYPE SPECIFICATION AND OTHER WORKFLOW CLASSES 30
TABLE 2.19 SPECIFICATION OF THE PARTICIPANT ASSIGNMENT ATTRIBUTES 30
TABLE 2.20 ASSOCIATIONS BETWEEN PARTICIPANT ASSIGNMENT AND OTHER WORKFLOW CLASSES 31
TABLE 2.21 SPECIFICATION OF THE APPLICATION CALL ATTRIBUTES 31
TABLE 2.22 ASSOCIATIONS BETWEEN APPLICATION CALL SPECIFICATION AND OTHER WORKFLOW CLASSES 31
TABLE 2.23 SPECIFICATION OF THE PROCESS INSTANCE ATTRIBUTES 32
TABLE 2.24 ASSOCIATIONS BETWEEN PROCESS INSTANCE AND OTHER WORKFLOW CLASSES 34
TABLE 2.25 PROCESS INSTANCE STATE DESCRIPTION 35
TABLE 2.26 SPECIFICATION OF THE PROCESS INSTANCE STATE ATTRIBUTES 35
TABLE 2.27 ASSOCIATIONS BETWEEN PROCESS INSTANCE STATE AND OTHER WORKFLOW CLASSES 36
TABLE 2.28 ASSOCIATIONS BETWEEN CONTAINER ATTRIBUTE AND OTHER WORKFLOW CLASSES 36
TABLE 2.29 SPECIFICATION OF THE ACTIVITY INSTANCE ATTRIBUTES 36
TABLE 2.30 ASSOCIATIONS BETWEEN ACTIVITY INSTANCE AND OTHER WORKFLOW CLASSES 37
TABLE 2.31 ACTIVITY INSTANCE STATE DESCRIPTION 39
TABLE 2.32 SPECIFICATION OF THE ACTIVITY INSTANCE STATE ATTRIBUTES 40
TABLE 2.33 ASSOCIATIONS BETWEEN ACTIVITY INSTANCE STATE AND OTHER WORKFLOW CLASSES 41
TABLE 2.34 SPECIFICATION OF THE TRANSITION INSTANCE ATTRIBUTES 41
TABLE 2.35 ASSOCIATIONS BETWEEN TRANSITION INSTANCE AND OTHER WORKFLOW CLASSES 41
TABLE 2.36 SPECIFICATION OF THE OBJECT REFERENCE ATTRIBUTES 41
TABLE 2.37 ASSOCIATIONS BETWEEN OBJECT REFERENCE AND OTHER WORKFLOW CLASSES 42

TABLE 2.38 SPECIFICATION OF THE OBJECT REFERENCE ATTRIBUTES 42
TABLE 2.39 ASSOCIATIONS BETWEEN OBJECT REFERENCE AND OTHER WORKFLOW CLASSES 42
TABLE 2.40 SPECIFICATION OF THE APPLICATION CALL ATTRIBUTES 43
TABLE 2.41 ASSOCIATIONS BETWEEN APPLICATION CALL AND OTHER WORKFLOW CLASSES 43
TABLE 2.42 SPECIFICATION OF THE WORKFLOW ATTRIBUTE DEFINITION ATTRIBUTES 43
TABLE 2.43 SPECIFICATION OF THE WORKFLOW ATTRIBUTE ATTRIBUTES 44
TABLE 2.44 SPECIFICATION OF THE (ADDITIONAL) PARAMETER DEFINITION ATTRIBUTES 45
TABLE 2.45 ASSOCIATIONS BETWEEN PARAMETER DEFINITION AND OTHER WORKFLOW OBJECTS 45
TABLE 2.46 SPECIFICATION OF THE (ADDITIONAL) PARAMETER ATTRIBUTES 46
TABLE 2.47 ASSOCIATIONS BETWEEN PARAMETER AND OTHER WORKFLOW OBJECTS 46
TABLE 2.48 SPECIFICATION OF THE WORKFLOW EXPRESSION ATTRIBUTES 46
TABLE 2.49 SPECIFICATION OF THE WORKFLOW EXPRESSION EVALUATION ATTRIBUTES 47
TABLE 3.1 COMPARISON OF THE STANDARD QUERY LANGUAGES 70
TABLE 5.1 SPECIFICATION OF THE BUILT-IN WORKFLOW MONITORING FUNCTIONS 101