ODRA: A Next Generation Object-Oriented Environment for Rapid Database Application Development

Michał Lentner and **Kazimierz Subieta**
Polish-Japanese Institute of Information Technology
ul. Koszykowa 86, 02-008 Warszawa, Poland
{m.lentner, subieta}@pjwstk.edu.pl

Abstract. ODRA (Object Database for Rapid Application development) is an object-oriented application development environment currently being constructed at the Polish-Japanese Institute of Information Technology. The aim of the project is to design a next-generation development tool for future database application programmers. The tool is based on the query language SBQL (Stack-Based Query Language), a new, powerful and high level object-oriented programming language tightly coupled with query capabilities. The SBQL execution environment consists of a virtual machine, a main memory DBMS and an infrastructure supporting distributed computing. The paper presents design goals of ODRA, its fundamental mechanisms and some relationships with other solutions.

1 Introduction

With the growth of non-classical database application, especially in the rapidly growing Internet context, the issue of bulk data processing in distributed and heterogeneous environments is becoming more and more important. Currently, increasing complexity and heterogeneity of this kind of software has led to situation where programmers are no more able to grasp every concept necessary to produce applications that could efficiently work in such environments. The number of technologies, APIs, DBMSs, languages, tools, servers, etc. which the database programmer should learn and use is extremely huge. This results in enormous software complexity, extensive costs and time of software manufacturing and permanently growing software maintenance overhead. Therefore the research on new, simple, universal and homogeneous ideas of software development tools is currently very essential.

The main goal of the ODRA project is to develop new paradigms of database application development. We are going to reach this goal by increasing the level of abstraction at which the programmer works. To this end we introduce a new, universal, declarative programming language, together with its distributed, database-oriented and object-oriented execution environment. We believe that such an approach provides functionality common to the variety of popular technologies (such as relational/object databases, several types of middleware, general purpose programming languages and their execution environments) in a single universal, easy to learn, interoperable and effective to use application programming environment.

The principle ideas which we are implementing in order to achieve this goal are the following:

- **Object-oriented design.** Despite the principal role of object-oriented ideas in software modeling and in programming languages, these ideas have not succeeded yet in the field of databases. As we show in this paper, our approach is different from current ways of perceiving object databases, represented mostly by the ODMG standard [4] and database-related Java technologies (e.g. [5, 6]). Instead, we are building our system upon a methodology called the Stack-Based Approach (SBA) to database query and programming languages [13, 14]. This allows us to introduce for database programming all the popular object-oriented mechanisms (like objects, classes, inheritance, polymorphism, encapsulation), as well as some mechanisms previously unknown (like dynamic object roles [1,7] or interfaces based on database views [8, 10]).

- **Powerful query language extended to a programming language.** The most important feature of ODRA is SBQL (Stack-Based Query Language), an object-oriented query and programming language. SBQL differs from programming languages and from well-known query languages, because it is a query language with the full computational power of programming languages. SBQL alone makes it possible to create fully fledged database-oriented applications. The possibility to use the same very-high-level language for most database application development tasks may greatly improve programmers' efficiency, as well as software stability, performance and maintenance potential.

- **Virtual repository as middleware.** In a networked environment it is possible to connect several hosts running ODRA. All systems tied in this manner can share resources in a heterogeneous and dynamically changing, but reliable and secure environment. Our approach to distributed computing is based on object-oriented virtual updatable database views [9]. Views are used as *wrappers* (or mediators) on top of local servers, as a *data integration* facility for global applications, and as *customizers* that adopt global resources to needs of particular client applications. This technology can be perceived as contribution to distributed databases, Enterprise Application Integration (EAI), Grid Computing and Peer-To-Peer networks.

The rest of the paper is organized as follows. In Section 2 we shortly present the main motivations and features of SBQL. In Section 3 we discuss application integration using ODRA. In Section 4 we discuss various deployment scenarios concerning ODRA-based applications. Section 5 concludes.

2 SBQL

The term *impedance mismatch* denotes a well-known infamous problem with mapping data between programming languages (recently Java) and databases. Majority of Java programmers spend between 25% and 40% of their time trying to map objects to relational tables and v/v. In order to reduce the negative influence of the feature, some automatic binding mechanisms between programming language objects and database structures have been suggested. This approach is expressed in the ODMG standard, post-relational DBMS-s and several Java technologies.

Unfortunately, all these solutions have only shown that despite the strong alignment of the database constructs with the data model of the programming languages used to manipulate them, the impedance mismatch persists. It could be reduced at the cost of giving up the support for a

higher-level query language, which is unacceptable. Impedance mismatch is a bunch of negative features emerging as a result of too loose coupling of a query language and a general-purpose programming language. The incompatibilities concern syntax, type checking, language semantics and paradigms, levels of abstraction, binding mechanisms, namespaces, scope rules, iteration schemas, data models, ways of dealing with such concepts as null values, persistence, generic programming, etc. The incompatibilities cannot be resolved by any, event apparently reasonable approach to modify functionality or additional utilities of existing programming languages.

All these problems could be completely eliminated by means of a new self-contained query/programming language based on homogeneous concepts. Our idea concerns an imperative object-oriented programming language, in which there is no distinction between expressions and queries. Such expressions/queries should have features common to traditional programming language expressions (literals, names, operators), but also should allow for declarative data processing. Query operators can be freely combined with other language's constructs, including imperative operators, control statements and programming abstractions. The language could be used not only for application programming, but also to query/modify databases stored on disk and in main memory. Our proposal of such a language is named SBQL and it is the core of ODRA.

2.1 Queries as Expressions

SBQL is defined for a very general data store model, based on the principles of object relativism and internal identification. Each object has the following properties: an internal identifier, an external name and some value. There are three kinds of objects:

- simple (<OID, name, atomic value>),
- complex (<OID, name, set of subobjects>),
- reference (<OID, name, target OID>).

There are no dangling pointers: if a referenced object is deleted, the reference objects that point at it, are automatically deleted too. There are no null values - lack of data is not recorded in any way (just like in XML). This basic data model can be used to represent relational and XML data. We can also use it to build more advanced data structures which are properties of more complex object-oriented data models (supporting procedures, classes, modules, etc).

SBQL treats queries in the same way as traditional programming languages deal with expressions (we therefore use the terms query and expression interchangeably). Basic queries are literals and names. More complex queries are constructed by connecting literals and names with operators. Thus, every query consists of several subqueries and there are no limitations concerning query nesting. Because of the new query nesting paradigm, we avoid the *select-from-where* sugar, which is typical for SQL-like query languages. In SBQL expressions/queries are written in the style of programming languages, e.g. 3+1; $(x+y)*z$; *Employee* **where** *salary* > $(x+y)$; etc.

In SBQL we have at least six query result kinds: atomic values, references, structures, bags, sequences and binders. They can be combined and nested in fully orthogonal way (limited only by type constraints). Unlike ODMG, we do not introduce explicitly collections (bags, sequences) in the data store; collections are substituted by many objects with the same name (like in XML). Structures are lists of fields (possibly unnamed) which are together treated as a single value. Binders are pairs <*name, result*>, written as *name*(*result*), where *result* is any query result. Query results are not objects - they have no OIDS and may have no names. Query results and procedure parameters belong to the same domain, hence it is possible to pass a query as a parameter. SBQL

queries never return objects, but references to them (OIDs). Due to this feature not only *call-by-value*, but also *call-by-reference* and other parameter passing styles are possible.

It is possible to create a procedure which returns a collection of values. Such a procedure reminds a database view, but it can encapsulate complex processing rather than a single query (like in SQL). However, in SBQL procedures and (updateable) views are different abstractions; the second one has no precedents in known query and programming languages. Names occurring in queries are bound using the environment stack, which is a structure common to most programming languages. Its sections are filled in with binders. Stack sections appear not only as results of procedure calls, but also due to a specific group of query operators, called *non-algebraic*. Among them there are such operators as: . (dot), *where*, *join*, *order by*, *forall*, *forany*, etc. All non-algebraic operators are macroscopic, i.e. work on collections of data.

Query operators called *algebraic* do not use the environment stack. Some of them (e.g. *avg*, *union*) are macroscopic, some other (e.g. +, -, *) are not. Algebraic and non-algebraic operators, together with the environment stack and the query result stack make up typical functionality expected from a query language designed to deal with structured and semi-structured data.

Some SBQL operators provide functionality which is absent or very limited in popular query languages. Among them are transitive closures and fixed-point equations. Together with procedures (which can be recursive), they constitute three styles of recursive programming in SBQL. All SBQL operators are fully orthogonal with imperative constructs of the language.

SBQL queries can be optimized using methods known from programming languages and databases. SBQL optimizer supports well known techniques, such as rewriting, cost-based optimization, utilization of indices and features of distributive operators (like shifting selection before join). Several powerful strategies unknown in other query languages (like shifting independent subqueries before non-algebraic operators) have been developed [11, 13]. The process of optimization usually occurs on the client-side during the program compilation process. The traditional server-side optimization is also possible. Since the SBQL compiler provides static counterparts of runtime mechanisms (database schema, static environment stack and static result stack), it is possible to use these counterparts to simulate the whole program evaluation process during compile time. This makes it possible strong and semi-strong type checking of SBQL queries/programs [12] and compile-time optimizations.

Thanks to the data independence, global declarations are not tied with programs. Instead, they are parts of a database schema and can be designed, administered and maintained independently of programs. If necessary, a client-side SBQL compiler automatically downloads the metabase (which contains the database schema, database statistics, and other data) from the server to accomplish strong type checking and query optimization.

2.2 Advanced SBQL Features

SBQL supports popular imperative programming language constructs and mechanisms. There are well known control structures (if, loop, etc.), as well as procedures, classes, interfaces, modules, and other programming and database abstractions. All are fully orthogonal with SBQL expressions. Most SBQL abstractions are first-class citizens, which means they can be created, modified, deleted and analyzed at runtime. Declarations of variables in SBQL may determine many objects of the same name and type. Thus, the concept of variable declaration is similar to table creation in relational databases.

Apart from variable name and type, variable declarations determine cardinalities, usually [0..*], [1..*], [0..1] and [1..1]. A cardinality constraint assigned to a variable is the way in which SBQL treats collections. It is possible to specify whether variable name binding should return ordered (sequences) or unordered (bags) collections. Arrays are supported as ordered collections with fixed length.

In classical object-oriented languages (e.g. Java) types are represented in a relatively straightforward manner and suitable type equivalence algorithms are not hard to specify and implement. However, type systems designed for query languages have to face such problems as irregularities of data structures, repeating data (collections with various cardinality constraints), ellipses, automatic coercions, associations among objects, etc. These peculiarities of query languages make existing approaches to types too limited. SBQL provides a semi-strong type system [12] with a relatively small set of types and with structural type conformance. SBQL is perhaps the only advanced query language with the capability of static type checking. Static type checking in SBQL is based on the mechanisms used also during static optimization (described above): the static environment stack, the static query result stack, and the metabase (which contains variable declarations). Note that traditional query processing assumes that queries are embedded in a host language as strings, which makes static type checking impossible.

SBQL programs are encapsulated within modules, which constraint access from/to their internals through import and export lists. Modules are considered complex objects which may contain other objects. Because modules are first-class citizens, their content may change during run-time. Classes are complex objects too consisting of procedures and perhaps other objects. Procedures stored inside classes (aka *methods*) differ from regular procedures only by their execution environment, which additionally contains internals of a currently processed object.

Procedure parameters and results can be bulk values. All procedure parameters (even complex ones) can be passed by values and by references. Due to the stack-based semantics procedures can be recursively called with no limitations and with no special declarations.

SBQL supports two forms of inheritance: class inheritance (static) and object inheritance (dynamic). The former is typical of popular programming languages. Unlike Java, multiple inheritance is allowed. The second inheritance form is also known as dynamic object roles [1, 7]. This concept assumes that throughout its lifetime an object can gain and lose multiple roles. For example, the object Person can have simultaneously such roles as Employee, Student, Customer, etc. Roles may be dynamically inserted or deleted into/from an object. An object representing a role inherits all features of its super-objects. This method models real-life scenarios better than multiple inheritance. For instance, persons can be students, employees or customers only for some time of their entire life. The mechanism of dynamic object roles solves many problems and contradictions, in particular, with multiple inheritance, tangled aspects, historical objects, etc.

Many SBQL concepts have their roots in databases rather than in programming languages. One of them is the mechanism of updatable views [8, 10]. Such views not only present the content of a database in different ways, but also allow to perform update operations on virtual data in a completely transparent way. In SBQL a view definition contains a procedure generating so-called *seeds* of virtual objects. The definition also bears specification of procedures that are to be performed on stored objects in response to update operations addressing virtual objects generated by the view. The procedures overload generic operations (create, retrieve, update, insert, delete) performed on virtual objects. The view definer is responsible for implementing every operation

that must be performed after a virtual object is updated. There are no restrictions concerning which operations on virtual objects are allowed and which are not (unlike view updateability criteria known from other proposals). Because a view definition is a complex object, it may contain other objects, in particular, nested view definitions, ordinary procedures, variables (for stateful views), etc.

Updatable views in ODRA have many applications, ranging from traditional (virtual mapping of data stored in the database) to complete novelty (mediators or integrators). In particular, since SBQL can handle semi-structured data, SBQL views can be used as an extremely powerful transformation engine (instead of XSLT). The power of SBQL views is the power of a universal programming language, concerning both the mapping of stored objects into virtual ones and the mapping of a updates of virtual objects into updates of stored ones.

Another case is the concept of the interface, which is also expressed as a view. Because an interface is a first-class citizen, apart from tasks common to traditional interfaces, it can also serve as an element of the security subsystem. A more privileged user has access to more data inside an object, while another user sees its internals through a separate, limited interface.

2.3 SBQL Runtime Environment

An SBQL program is not directly compiled into a machine code. It is necessary to have some intermediate forms of programs and a virtual execution environment which executes them. The first form is a syntactic tree, which is the subject of optimizations and type checking. This form is transformed into a bytecode (different from a Java bytecode, for several important reasons). The SBQL execution environment consists of a virtual machine (VM) acting on a bytecode. The VM functionality provides services typical for hardware (virtual instruction set, virtual memory, etc.) and operating systems (loading, security, scheduling, etc.). Once compiled, a bytecode can be run on every system for which ODRA has been ported. We plan that SBQL programs can also move from one computer to another during runtime (e.g. from a busy computer to an idle one).

The DBMS part controls the data store and provides such mechanisms as transaction support, indexing, persistence, etc. It is a main memory DBMS, in which persistence is based on modern operating systems' capabilities, such as memory mapped files. ODRA assumes the orthogonal persistence model [2].

3 Application Integration Using ODRA

The distributed nature of contemporary information systems requires highly specialized software facilitating communication and interoperability between applications in a networked environment. Such software is usually referred to as middleware and is used for application integration. ODRA supports information-oriented and service-oriented application integration. The integration can be achieved through several techniques known from research on distributed/federated databases. The key feature of ODRA-based middleware is the concept of transparency. Due to transparency many complex technical details of the distributed data/service environment need not to be taken into account in the application code. ODRA supports such transparency forms as transparency of updating made from the side of a global client, transparency of distribution and heterogeneity, transparency of data fragmentation, transparency of data/service redundancies and replications, transparency of indexing, etc.

These forms of transparency have not been solved to a satisfactory degree by current technologies. For example, Web Services support only transparency of location and transparency of implementation. Transparency is achieved in ODRA through the concept of a virtual repository (Fig. 1). The repository seamlessly integrates distributed resources and provides a global view on the whole system, allowing one to utilize distributed software resources (e.g. databases, services, applications) and hardware (processor speed, disk space, network, etc.). It is responsible for the global administration and security infrastructure, global transaction processing, communication mechanisms, ontology and metadata management. The repository also facilitates access to data by several redundant data structures (global indexes, global caches, replicas), and protects data against random system failures.
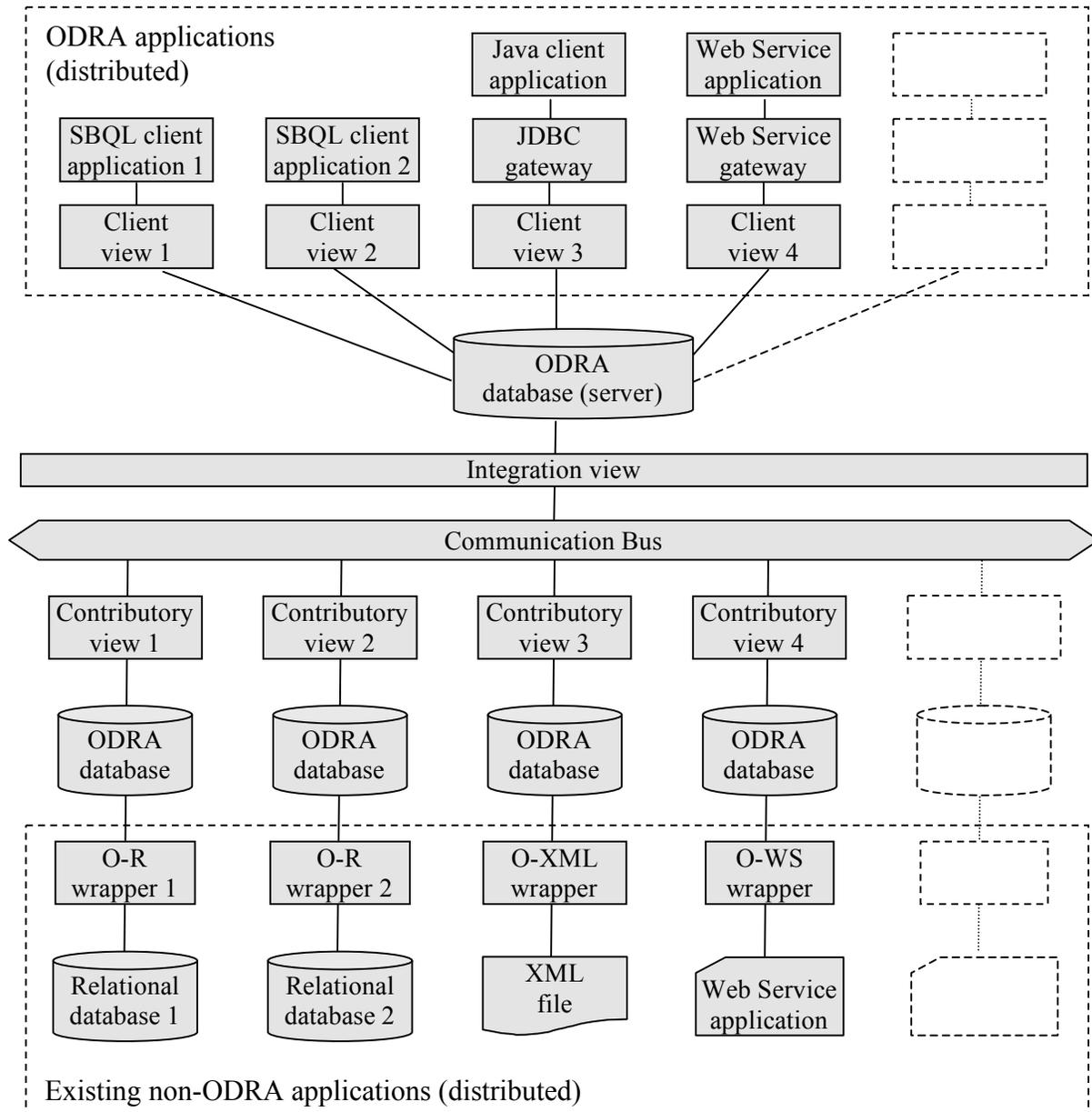
Fig.1. ODRA Virtual Repository Architecture

The user of the repository sees data exposed by the systems integrated by means of the virtual repository through the global integration view. The main role of the integration view is to hide complexities of mechanisms involved in access to local data sources. The view implements CRUD behavior which can be augmented with logic responsible for dealing with horizontal and vertical fragmentation, replication, network failures, etc. Thanks to the declarative nature of SBQL, these complex mechanisms can often be expressed in one line of code. The repository has a highly decentralized architecture. In order to get access to the integration view, clients do not send queries to any centralized location in the network. Instead, every client possesses its own copy of the global view, which is automatically downloaded from the integration server after successful authentication to the repository. A query executed on the integration view is to be optimized using such techniques as rewriting, pipelining, global indexing and global caching.

Local sites are fully autonomous, which means it is not necessary to change them in order to make their content visible to the global user of the repository. Their content is visible to global clients through a set of contributory views which must conform to the integration view (be a subset of it). Non-ODRA data sources are available to global clients through a set of wrappers, which map data stored in them to the canonical object model assumed for ODRA. We are developing wrappers for several popular databases, languages and middleware technologies. Despite of their diversity, they can all be made available to global users of the repository. The global user may not only query local data sources, but also update their content using SBQL. Instead of exposing raw data, the repository designer may decide to expose only procedures. Calls to such procedures can be executed synchronously and asynchronously. Together with SBQL's support for semi-structured data, this feature enables document-oriented interaction, which is characteristic of current technologies supporting Service Oriented Architecture (SOA).

4 Deployment Scenarios

ODRA is a flexible system which can act as a substitution of many current technologies. In particular, it can be used to build:

- **Standalone applications.** There are several benefits for programmers who want to create applications running under ODRA. Since SBQL delivers a set of mechanisms allowing one to use declarative constructs, programming in SBQL is more convenient than in languages usually used to create business applications (such as Java). ODRA transparently provides such services as persistence; thus applications do not have to use additional DBMS-s for small and medium data sets. Other databases can be made visible through a system of wrappers, which transparently map their content to the canonical data model of ODRA.

- **Database systems**. ODRA can be used to create a traditional client-server database system. In this case, one installation of ODRA plays the role of a database server, other act as clients. A client can be an application written in SBQL and can run in ODRA, as well as a legacy application connected by a standard database middleware (e.g. JDBC). The system may be used as a DBMS designed to manage relational, object-oriented and XML-oriented data. In every case all database operations (querying, updating, transforming, type checking, etc.) can be accomplished using SBQL. The technologies, such as SQL, OQL, XQuery [15], XSLT, XML Schema, can be fully substituted by the SBQL capabilities.

- **Object request brokers**. ODRA-based middleware is able to provide functionality known from distributed objects technologies (e.g. CORBA). In ODRA, the global integrator provides a global view on the whole distributed system, and the communication protocol transports requests and responses between distributed objects. Local applications can be written in any programming language, as the system of wrappers maps local data to ODRA's canonical model. There are several advantages of ODRA comparing to traditional ORB technologies. Firstly, the middleware is defined using a query language, which speeds up middleware development and facilitates its maintenance. Secondly, in CORBA it is assumed that resources are only horizontally partitioned, not replicated and not redundant. ODRA supports horizontal and vertical fragmentation, can resolve replications and can make choice from redundant or replicated data.

- **Application servers.** A particular installation of ODRA can be chosen to store application logic. By doing so, the developers can exert increased control over the application logic through centralization. The application server can also take several existing enterprise systems, map them into ODRA's canonical model and expose them through a Web-based user interface. It is possible to specify exactly how such an application server behaves, so the developer can implement such mechanisms as clustering, or load balancing by him/her own using declarative constructs or already implemented components.

- **Integration servers**. Integration servers can facilitate information movement between two or more resources, and can account for differences in application semantics and platforms. Apart from that, integration servers provide such mechanisms as: message transformation, content based routing, rules processing, message warehousing, directory services, repository services, etc. The most advanced incarnation of this technology (called Enterprise Service Bus, ESB) is a highly decentralized architecture combining concepts of Message Oriented Middleware (MOM), XML, Web Services and workflow technologies. Again, this technology is one of the forms that our updatable view-based middleware can take.

- **Grid Computing infrastructure**. Grid Computing is a technology presented by its advocates as integration of many computers into one big virtual computer, which combines all the resources that particular computers possess. People involved in grid research usually think of resources in terms of hardware (computation, storage, communications, etc.), not data. It is a result of their belief that "in a grid, the member machines are configured to execute programs rather than just move data" [3]. However, all business applications are data-intensive and in our opinion distribution of computation depends almost always on data location. Moreover, responsibility, reliability, security and complexity of business applications imply that distribution of data must be a planned phase of a disciplined design process. ODRA supports this point of view and provides mechanisms enabling grid technology for businesses.

5 Conclusion

We have presented an overview of the system ODRA, which is used as our research platform aiming future database application development tools. ODRA comprises a very-high-level object-oriented query/programming language SBQL, its runtime environment integrated with a DBMS, and a novel infrastructure designed to integration of distributed applications. The core of the prototype is already operational and is used experimentally for various test cases. The project is still under way and focuses on adding new functionalities, improving existing ones,

implementing new interoperability modules (wrappers), implementing specialized network protocols and elaborating various optimization techniques.

References

[1]   A.Albano, R.Bergamini, G.Ghelli, R.Orsini. An Object Data Model with Roles. Proc. VLDB Conf., 39-51, 1993

[2]   M.Atkinson, R.Morrison. Orthogonally Persistent Object Systems. The VLDB Journal 4(3), 319-401, 1995

[3]   V.Berstis: Fundamentals of Grid Computing. IBM Redbooks paper. IBM Corp. (2002) http://www.redbooks.ibm.com/redpapers/pdfs/redp3613.pdf.

[4]   R.G.G.Cattell, D.K.Barry (Eds.): The Object Data Standard: ODMG 3.0. Morgan Kaufmann 2000.

[5]   W.R.Cook, C.Rosenberger: Native Queries for Persistent Objects A Design White Paper. http://www.db4o.com/about/productinformation/whitepapers/Native%20Queries%20White paper.pdf, 2006

[6]   Hibernate - Relational Persistence for Java and .NET. http://www.hibernate.org/, 2006

[7]   A.Jodlowski, P.Habela, J.Plodzien, K.Subieta: Objects and Roles in the Stack-Based Approach. Proc. DEXA Conf., Springer LNCS 2453, 2002.

[8]   H.Kozankiewicz: Updateable Object Views. PhD Thesis, 2005, http://www.ipipan.waw.pl/ ~subieta/ -> Finished PhD-s -> Hanna Kozankiewicz

[9]   H.Kozankiewicz, K.Stencel, K.Subieta: Integration of Heterogeneous Resources through Updatable Views. Workshop on Emerging Technologies for Next Generation GRID (ETNGRID-2004), June 2004, Proc. published by IEEE.

[10]   H.Kozankiewicz, J.Leszczylowski, K.Subieta: Updateable XML Views. Proc. of ADBIS'03, Springer LNCS 2798, 2003, 385-399.

[11]   J.Plodzien, A.Kraken: Object Query Optimization in the Stack-Based Approach. Proc. ADBIS Conf., Springer LNCS 1691, 3003-316, 1999.

[12]   K.Stencel: Semi-strong Type Checking in Database Programming Languages, PJIIT - Publishing House, ISBN 83-89244-50-0, 2006, 207 pages (in Polish).

[13]   K.Subieta. Theory and Construction of Object-Oriented Query Languages. PJIIT - Publishing House, ISBN 83-89244-28-4, 2004, 522 pages (in Polish).

[14]   K.Subieta: Stack-Based Approach (SBA) and Stack-Based Query Language (SBQL). http://www.sbql.pl, 2006

[15]   World Wide Web Consortium (W3): XML Query specifications. http://www.w3.org/XML/Query/