Odra: a next–generation object–oriented environment for rapid database application development. Motivation, general assumptions and architecture.

Michał Lentner and Jan Kowalski

Polish–Japanese Institute of Information Technology
ul. Koszykowa 86
02–008 Warszawa, Poland
{m.lentner, j.kowalski}@pjwstk.edu.pl

Abstract. Odra is an 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. We claim that such systems should be based on new, high level programming languages tightly coupled with query processing capabilities. Our system is therefore based on a powerful query language called SBQL and its execution environment consisting of a virtual machine, a main memory database management system and infrastructure supporting distributed computing. The design goals of Odra, its fundamental mechanisms, as well as its comparison with existing solutions are presented in this paper.

1 Introduction

With the growth of the Internet, the issue of bulk data processing in distributed and heterogeneous environments is becoming more and more important. Correspondingly, the tools that facilitate this kind of activity are more and more desirable. On the other hand, the increasing complexity and heterogeneity of this kind of software implies a situation where programmers are no more able to grasp every concept necessary to produce applications that could efficiently work in such environments. Currently, the amount of technologies, APIs, languages, servers, etc. which the database programmer should learn and use is huge. The growing amount of concepts results in gigantic software complexity, which means colossal costs of its development and maintenance. This problem will soon be even more severe, therefore research on new, simple, universal and homogeneous ideas of software development tools is essential.

The main goal of project Odra 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 execution environment. We believe that our approach provides functionality common to popular technologies (such as relational/object databases, several types of middleware, general purpose programming languages and their execution environments) in one, consistent, universal, easy to learn, interoperable and effective to use, application programming environment.

Let us begin by outlining the principle ideas which we are implementing in order to achieve our goals. They are:

- *Object oriented design.* Despite the principal role of object–oriented ideas in programming languages, these ideas have not succeeded in the field of databases. As we will show in this paper, our approach is different from current ways of perceiving object databases, represented mostly by the ODMG standard and ODMG–related Java technologies. Instead, we are building our system upon a theory called the stack based approach to query languages (SBA) [1]. This allows us to introduce for databases all the popular object–oriented mechanisms (like objects, classes, inheritance, polimorphism, hermetization), as well as several mechanisms previously unknown (like dynamic object roles, or interfaces based on database views).
- *Powerful query language extended to a programming language.* The most important element of Odra is a language called SBQL. SBQL differs considerably from ordinary programming languages because it is a query language. It is, however, different from well known query languages, because it has fully computational power. SBQL allows us to query/modify relational databases, XML repositories, object databases, as well as create fully fledged database–oriented applications. The possibility to use the same language for most database application development tasks greatly improves programmers' efficiency, as well as software stability and performance.
- *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 updatable database views [5]. Views are used as wrappers/mediators on top of local servers, as a data integration facility for global applications. This technology can be perceived as our contribution to such areas of industry and science as distributed databases, Enterprise Application Integration (EAI) and Grid Computing.

2 SBQL

The term *impedance mismatch* represents a well know problem, usually described in the literature as a difficulty with mapping data between object–oriented programming languages (usually Java) and databases. It is a daunting problem, as most programmers spend between 25% and 40% of their time trying to map objects to relational tables.

In order to reduce the negative influence of this problem, some automatic binding between programming language objects and database structures has

been suggested. This approach is expressed in the ODMG standard, most post–relational database management systems and several Java technologies. Unfortunately, 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 would be unacceptable.

We tackle the problem of impedance mismatch starting with a much wider definition. In our opinion impedance mismatch is a set of detrimental features emerging as a result of formal unification between a query language and a general–purpose programming language. It results in several incompatibilities in the domains of syntax, type systems, language semantics and paradigms, levels of abstraction, binding phases and mechanisms, namespaces, scope rules, iteration schemas, data models, ways of dealing with such concepts as null values, persistence, generic programming, etc.

All these problems could be completely eliminated by means of a single, new, self–dependent query/programming language. It would be an imperative programming language, in which expressions would be treated as queries. Such expressions would have features common to traditional programming language expressions (literals, names, operators) and would be processed according to traditional rules (naming–scoping–binding). Expressions of this hypothetical language would allow for declarative data processing, but at the same time would be completely orthogonal with other language constructs. The language could be used not only for application programming, but also to query/modify databases stored on disk and in main memory.

We have designed such a language. Its name is SBQL, it is the core of Odra and we present its main features in this section of the paper.

2.1 Queries as expressions

SBQL is defined for a very general data store, 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, OIDs 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. 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 elaborate constructs which are characteristic 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 can be written using only two components: 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. There is no `SELECT..FROM..WHERE` syntactic sugar which is typical of SQL–like query languages. In order to add one to three in SBQL, we write `3 + 1`, not `SELECT 3 + 1 FROM DUAL`.

Traditional programming languages have expressions which can result in three kinds of values: atomic values, references and null. In SBQL we have as many as six query result types: atomic values, references, structures, bags, sequences, binders. Bags and sequences represent collections. We do not introduce collections in the data store (like in ODMG). Structures are lists of fields (possibly unnamed) which are together treated as a single value. Binders are single values composed of pairs <name, value>. Query results are not objects — they neither have names, nor OIDs. Because query results and procedure parameters belong to the same domain, it is possible to pass data between them. Queries never return objects, only references to them. Thanks to this feature not only call–by–value, but also call–by–reference and other parameter communication styles are possible. It is possible to create a procedure which returns a collection of values. Such a procedure can encapsulate complex logic, which is visible as a single, named expression (more than a view in relational databases).

Names are bound using the environment stack, which is a structure common to most programming languages. Its sections are filled with binders, which role is to make the process of binding possible. Stack sections appear not only as results of procedure calls, but also thanks to a specific group of operators, called non–algebraic. Among them are such operators as: **.**, **where**, **join**, **order by**, **for all**, **for any**, etc. All non–algebraic operators are macroscopic, which means they can work on collections of data. Algebraic operators do not use the environment stack. Some of them (e.g. **avg**, **union**) are macroscopic, some (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 advanced SBQL operators provide functionality which is absent or very limited in popular query languages. Among them are operators designed to compute transitive closures and fixpoint equations. Together with procedures (which obviously can be recursively called), they constitute three styles of recursive programming available in SBQL: programming languages style, algebraic style, and logic style. All SBQL operators are fully orthogonal with imperative constructs of the language. Such constructs are usually neglected by query language designers and are absent in many new languages.

SBQL queries can be optimized using methods known from programming languages and databases. Its optimizers support well known techniques, such as elimination of dead subqueries, cost–based optimization, utilization of indexes 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 also been developed [4]. The process of optimization usually occurs on the client side (but traditional, server–side optimization is also possible), during the program compilation process. Since SBQL compilers provide static counterparts of such runtime mechanisms, as the

object store, the environment stack and the result stack, it is possible to use these counterparts in order to simulate the whole program evaluation process. Thanks to the data independence characteristic of databases, global declarations are not tied with programs. Instead, they are parts of the 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 the process of 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, etc. All these structures are fully orthogonal with SBQL expressions. Most SBQL constructs are first–class citizens, which means they can be created, modified, deleted and analyzed at runtime.

Variables in SBQL may represent 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 bear some cardinality ([0..*], [1..*], [0..1], [1..1]). It is also 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 typical 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 (several kinds of collections with various cardinality constraints), ellipses, automatic coercions, recursion in type definitions, etc. These peculiarities of query languages make existing approaches to types inadequate.

SBQL can deal with such problems. The language provides a semi–strong type system with a relatively small set of types, and with structural type conformance. SBQL is probably the only 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 languages assume that queries are embedded in the host language source code as strings, which makes static type checking impossible to achieve.

SBQL programs are encapsulated in modules, which protect access to their internals using import and export lists. Modules are complex objects and may contain other objects. Because they are first–class citizens, their content may change over time. Classes are complex objects as well, and their internal structure also consists of other objects, i.e. procedures. As we have already mentioned, procedure parameters and results can represent many values. All procedure parameters (even complex ones) can be passed by values and by references. Pro-

cedures can be recursively called without any limitations, which is still a unique capability in the world of popular query languages.

The language supports two forms of inheritance: class inheritance and object inheritance. The former is typical of popular programming languages. Like in Java, only single inheritance is supported. The latter concept is supported by a mechanism called *dynamic object roles* [3]. This concept assumes that throughout its lifetime an object can gain and lose multiple roles. For example, the object `Person` can have such roles as `Employee`, `Student`, etc. An object which represents a role inherits all features of its super–objects. This method models real–life scenarios better than multiple inheritance does, since most persons are not employees/students throughout their entire life. The mechanism of dynamic object roles solves many problems associated with multiple inheritance, tangled aspects, temporal data, etc.

Many SBQL concepts have their roots in databases rather, than programming languages. One of them is the mechanism of updatable views [2]. 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 not only contains the query which generates virtual objects, but it also bears a specification of generic operations that are to be performed on stored objects in response to update operations executed on virtual objects generated by the view. Such a specification has the form of procedures that overload generic operations (create, retrieve, update, delete) performed on virtual objects. If any of those procedures is not specified, the particular operation is not allowed. In this way the view definer is responsible for implementing every operation that must be performed after a virtual object is updated. Thus, every virtual object can be modified, not only some of them. There are no special conditions concerning view updating. Because a view definition is a complex object, it may contain other objects: other view definitions (views can be nested), ordinary procedures, variables (stateful views), etc.

Updatable views in Odra have many applications, ranging from traditional (a specific representation of data stored in the database), to complete novelty. For example, since SBQL can handle semi–structured data, SBQL views can be used as a transformation engine (instead of XSLT). Another example is the concept of the interface, which is also expressed as a view. Because such 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

SBQL is a language which programs cannot be directly compiled into machine code. It is necessary to have some kind of an intermediate form of programs and a virtual execution environment which executes it. Odra provides an SBQL execution environment which comprises a virtual machine with an integrated database management system. The VM functionality provides services which

are typical of hardware (virtual instruction set, virtual memory, etc.) and of operating systems (loading, security, scheduling, etc.). Once compiled, SBQL code can be run on every system for which Odra has been ported. SBQL programs can also move from one computer to another during runtime (for example 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 database management system, in which persistence is based on modern operating systems' capabilities, such as memory mapped files.

3 Application integration using Odra

The distributed nature of contemporary information systems demands highly specialized software facilitating communication 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. It is an important idea, since thanks 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 forms of transparency as transparency of updating made from the side of a global client, transparency of 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 (e.g. databases, services, applications) and hardware (processor speed, disk space, network, etc.) resources. 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.

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. Every query executed on the integration view is carefully optimized using such techniques as pipelining, query decomposition, global indexing and global caching.

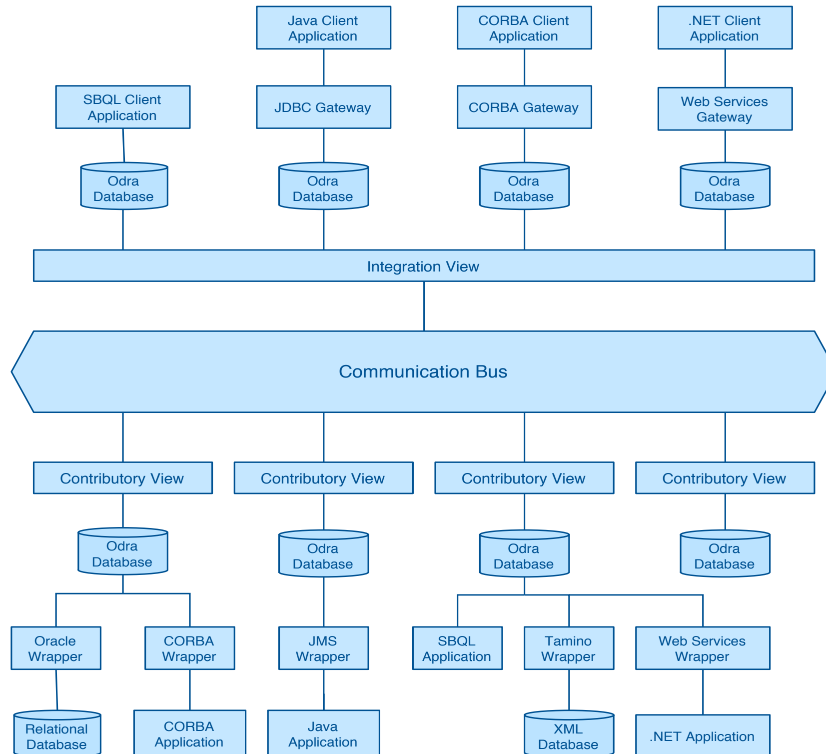

Fig. 1. Odra Virtual Repository Architecture

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 have developed 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 ordinary 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). Since Odra transparently provides such services as persistence, applications do not have to use additional database management systems 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 traditional client–server database systems. In this case, one installation of Odra plays the role of a database server, other act as clients. The 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 database management system designed to manage relational, object and XML data. In every case all database operations (querying, updating, transforming, type checking, etc.) can be realized using SBQL. Such technologies as SQL, OQL, XQuery, XSLT, XML Schema are not necessary.
– *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, 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, 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 mixing concepts of Message Oriented Middleware (MOM), XML, Web Services and Workflow technologies. Again, this technology is only 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" [6]. Unfortunately, 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 Summary and future work

In this paper we have presented an overview of system Odra, which is used as our research platform on future database application development tools. Odra comprises a very high level programming language, its runtime environment integrated with a DBMS, and a novel infrastructure designed to integration of federated databases. The core of the prototype is already operational and is used on a daily basis for application integration tasks. Work is still under way in the project mainly on improvements in optimization techniques.

References

1. K. Subieta, C. Beeri, F. Matthes, J.W. Schmidt: *A Stack–Based Approach to Query Languages.* Proc. East–West Database Workshop, 1994, Springer Workshops in Computing, 1995.
2. H. Kozankiewicz, J. Leszczyłowski, K. Subieta: *Updateable XML Views.* Proc. of ADBIS'03, Springer LNCS 2798, 2003, 385-399.
3. A. Jodłowski, P. Habela, J. Płodzien, K. Subieta: *Objects and Roles in the Stack-Based Approach.* Proc. DEXA Conf., Springer LNCS 2453, 2002.

4. J. Płodzien, A. Kraken: *Object Query Optimization in the Stack–Based Approach*. Proc. ADBIS Conf., Springer LNCS 1691, 3003-316, 1999.
5. 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.
6. V. Berstis: *Fundamentals of Grid Computing*. IBM Redbooks paper. IBM Corp. (2002).