Transparent Dynamic Database Evolution from Java

Awais Rashid, Peter Sawyer

Computing Department
Lancaster University
Lancaster
LA1 4YR
UK

marash@comp.lancs.ac.uk
sawyer@comp.lancs.ac.uk

ABSTRACT: *With the increasing provision by the ODMG standard and commercial OODBMS products for transparent access to the traditional database functionality of an OODBMS from Java, there is a need to provide Java programmers transparent access to advanced functionality such as dynamic evolution. This paper proposes a transparent API for the purpose. The API is based on a high-level object-oriented model. The application programmer interacts with the high-level model instead of interacting with the lower level model of the particular OODBMS. The various dynamic evolution features operate within the constraints of the Java type system.*

RÉSUMÉ:

KEY WORDS: Object Database Evolution, Transparent Evolution, Dynamic Evolution, Schema Evolution, Class Versioning, Type System Consistency

MOTS-CLÉS:

1. Introduction

The conceptual structure of an object-oriented database application may not remain constant and may vary to a large extent [SJO 93]. The need for these variations (evolution) arises due to a variety of reasons e.g. to correct mistakes in the database design or to reflect changes in the structure of the real world artefacts modelled in the database. Therefore, like any other database application object database applications are subject to evolution.

However, in object-oriented databases, support for evolution is a critical requirement since it is characteristic of complex applications for which they provide inherent support. These applications require dynamic modifications to both the data residing within the database and the way the data has been modelled i.e. both the objects residing within the database and the *schema* of the database are subject to change. Furthermore, there is a requirement to keep track of the change in case it needs to be reverted.

An object-oriented database management system needs not only traditional database functionality such as persistence, transaction management, recovery and querying facilities but also the ability to dynamically evolve, through various versions, both the objects and the class definitions. With the increasing provision by the ODMG standard [CAT 97] and commercial OODBMS products for transparent Java APIs, there is an urgent need to provide Java programmers with transparent access to advanced functionality such as dynamic evolution.

In this paper we present our database evolution system SADES which provides Java programmers transparent access to the dynamic evolution functionality of an OODBMS. SADES has been layered on top of the commercially available object database management system Jasmine[1] from Computer Associates and Fujitsu Limited.

The next section discusses the problem of achieving transparent object database evolution from Java due to its strong type system. Section 3 summarises the various evolution primitives to be supported by an OODBMS. The discussion is based on our earlier work [RAS 98a] [RAS 98b] [RAS 00a]. Section 4 describes the various evolution features offered by Jasmine in general and its Java bindings in particular. We argue that although Jasmine provides reasonable evolution features as compared to several other front-line object database management systems on the market, it does not support all the evolution primitives. Also, transparent access to evolution functionality from Java is limited. Section 5 describes the architecture of our database evolution system SADES and the transparent access to evolution functionality from Java. Section 6 summarises and concludes the paper.

[1] http://www.cai.com/jasmine

2. Transparent Evolution and the Java Type System

The seamless integration of object-oriented databases and Java [CAT 97] and the advent of orthogonal persistence for Java [ATK 96] have provided Java programmers transparent access to persistence functionality. However, extending this transparency to evolution features poses additional challenges due to strong differences among the underlying assumptions inherent in the fields of programming languages and database systems. A key requirement during evolution is to keep existing programs and data consistent with the semantics of the change. Due to the close integration of the database and the programming language data model in an OODBMS the constraints of the language type system which exist to improve safety act to hinder the required evolution.

An example scenario is the evolution of a class definition in Java. Let *class A* and *class B* be the definitions of the class before and after the evolution respectively. After evolution it may be desirable that all values of type *class A* now have *class B* as their type. However, such an operation is considered potentially dangerous by the Java type system (programs already bound to these values may rely on the assumption that they are of type *class A*) which prevents it.

The use of strongly typed linguistic reflection (as proposed by [KIR 96]) to achieve transparent evolution in the presence of the strong Java type system is not possible as Java reflection API is strictly read-only. Approaches such as [AMI 94] which detect unsafe statements at compile-time and check them at run-time using a specific clause automatically inserted around them can be employed. Such approaches support exceptions to behavioural consistency rules without sacrificing type safety. However, they are only usable for static type checking and, hence, not suitable during dynamic evolution. Any transparent, dynamic database evolution approach for Java must be able to operate within the constraints of the existing type system i.e. it must provide type safe dynamic evolution without:

— relying on a read-write reflective mechanism
— modifying the Java type system

3. Evolution Primitives

Historically, the database community has employed three fundamental techniques for modifying the conceptual structure of an object-oriented database, namely:

— schema evolution [BAN 87], where the database has one logical schema to which class definition and class hierarchy modifications are applied
— schema versioning [KIM 88] [LAU 97] [RA 97], which allows several versions of one logical schema to be created and manipulated independently of each other
— class versioning [MON 93] [SKA 86], which keeps different versions of each type and binds instances to a specific version of the type

In contrast to the above three approaches our approach [RAS 98a] [RAS 98b] superimposes schema evolution on class versioning and views conceptual database evolution as a composition of:

1. Class hierarchy evolution
2. Class versioning
3. Object versioning

Using the above evolution approach as a basis we have formulated an evolution taxonomy for object-oriented databases. The various evolution primitives in the taxonomy are listed below:

- Class Hierarchy Evolution
    - Add a new class
        - Add a class that forms a leaf node in the hierarchy graph
        - Add a class that forms a non-leaf node in the hierarchy graph
    - Drop an existing class
        - Drop a class that forms a leaf node in the hierarchy graph
        - Drop a class that forms a non-leaf node in the hierarchy graph
    - Modify DAG[2]-level inheritance relationships
        - Re-position an existing class in the hierarchy graph
        - Add an existing class to the super-class list of a class
        - Drop an existing class from the super-class list of a class
    - Rename a class
- Class Versioning
    - Derive a new class version
        - Add a new property
        - Add a new method
        - Drop an existing property
        - Drop an existing method
        - Modify the definition of a property
        - Modify the signature or body of a method
        - Change the super-class version
    - Remove an existing class version
        - Drop a version that forms a leaf-node in the class version derivation graph
        - Drop a version that forms a non-leaf node in the class version derivation graph
- Object Versioning
    - Derive a new version
        - Preserve a change in the values of the properties
        - Merge two or more existing versions
    - Delete an existing version
    - Change the status of a version
        - Derive a transient version from another transient version

[2] *Directed Acyclic Graph* for the class hierarchy

- Explicitly upgrade a transient version to the status of a working version
- Upgrade a working version to the status of a released version
- Downgrade a released version to the status of a working version

The taxonomy also mandates facilities for traversing the meta-object hierarchy and the structure of meta-objects. We recognise that the object versioning features listed above should be complemented with:

— facilities to check-in/check out objects to/from the database from/to private or group workspaces

— long transactions

— advanced locking mechanisms

However, these are beyond the scope of this paper.

4. Jasmine Evolution Features

The contents of this section are based on experiences with Jasmine 1.2 [CA 96] on Windows NT 4.0. The Jasmine OODBMS provides a wide range of features such as a multi-threaded database server and support for C/C++, Java, Active-X, Internet and multimedia applications. Although Jasmine is not ODMG compliant one of the Jasmine Java bindings is quite close to the ODMG Java binding and supports OQL, the ODMG Object Query Language.

Jasmine provides its own database language ODQL; Object Data Query Language. ODQL is an object oriented language and provides constructs for both data definition and manipulation in addition to querying facilities. ODQL is polymorphic in nature and supports multiple inheritance. ODQL statements can be entered interactively using an ODQL interpreter, embedded within a host language, constructed dynamically at run-time or executed through the C or Java API. Note that ODQL statements can only be embedded within C or C++ and not Java. It should be noted that although Jasmine maintains version histories internally there are no object or class versioning facilities available to the user through any of the APIs discussed below.

4.1. *Jasmine Java Bindings*

The Jasmine Java bindings aim to bridge the gap between the database language and the programming language by providing a single-language model for application development. Jasmine offers various facilities for developing database applications in Java. These include *persistent Java* (*pJ*), *Java Beans* (*Jb*), *Java proxies* (*Jp*) and *Java API* (*J API*) for Jasmine.

The persistent Java (pJ) binding is quite close to the ODMG 2.0 Java binding and provides the Java programmer transparent access to the database. Like the ODMG Java binding pJ uses a Java preprocessor which augments Java classes with the code required to achieve persistence and generates the corresponding database schema definitions. pJ also supports ODMG OQL besides ODQL. Achieving persistence through marking classes persistence capable at the pre-compilation or post-compilation stage is highly static in nature and does not leave room for dynamic schema modifications [RAS 99a]. pJ gets around the problem by providing access to the whole ODQL functionality. Member functions of system supplied classes can take as a parameter an ODQL statement and execute. It should, however, be noted that in the event that one chooses to do so, any mapping from ODQL classes to Java classes and vice versa becomes the application's responsibility. pJ, however, offers functionality to aid the mapping process.

Java beans (Jb) for Jasmine allow the development of Java applications using any Java Bean Development environment. Java proxies (Jp), on the other hand, allow Java applications to take advantage of the class libraries developed in ODQL by automatically generating Java classes statically bound to existing ODQL classes. The Java API (J API), in contrast to Java proxies, provides dynamic access to both Jasmine databases and their schema through Java. J API provides an implementation of the *DirContext* interface of the *Java Naming and Directory Interface* (*JNDI*)[3] in order to traverse the meta hierarchies in Jasmine databases. J API also offers facilities to add and drop class properties dynamically. Besides, it provides access to full ODQL functionality in a fashion similar to pJ hence allowing dynamic schema modifications if desired. Again correspondence between ODQL classes and Java classes in such a case has to be managed by the application.

The evaluation we presented in [RAS 99a] compares the evolution features offered by Jasmine and three other front-line object database management systems: POET[4], Versant[5] and O2[6]. The results clearly show that support for evolution functionality in Jasmine Java bindings is outstanding as compared to its three peers (see [RAS 99a]). However, only a subset of evolution primitives (listed in section 3) is supported. The features not supported are:

— Adding a class that forms a non-leaf node in the hierarchy graph
— Dropping a class that forms a non-leaf node in the hierarchy graph
— Modifying DAG-level inheritance relationships
— Class versioning
— Object versioning

We also observe that transparent access to evolution functionality from Java is limited to using J-API which offers functionality to:

— traverse the meta-object hierarchy
— traverse the structure of meta-objects
— add and drop class properties

The rest of the functionality has to be obtained using the ODQL gateways from pJ and J-API. ODQL provides facilities to access the database meta-data at run-time.

[3] http://java.sun.com/products/jndi/
[4] http://www.poet.com/
[5] http://www.versant.com/
[6] http://www.ardentsoftware.com/

Facilities are also available to dynamically add new classes and modify existing class definitions. New properties and methods can be added dynamically and existing ones removed or modified. Dynamic modifications to the class hierarchy are also possible. However, only a sub-class to an existing class can be added or removed at any time. Super-classes can be added or removed from a class only if it has not yet been validated.

The ODQL evolution functionality (through the ODQL gateways from pJ and J-API) is not available transparently to Java programmers as mapping between ODQL classes and Java classes has to be managed at the application level. ODQL is used as a declarative change specification language which results in impedance mismatch with Java.

5. SADES: Transparent Access to Evolution

We now describe our database evolution system SADES [RAS 98b] and the underlying database model which we presented in [RAS 98a]. SADES is being built as a layer on top of Jasmine. Traditional database functionality is obtained through Jasmine while dynamic evolution functionality is provided by SADES and is available transparently to Java programmers. SADES employs a composite active and passive knowledge-based approach to provide support for:

1. Class hierarchy evolution
2. Class versioning
3. Object versioning
4. Knowledge-base/rule-base evolution

The SADES conceptual schema [RAS 98a] is a fully connected directed acyclic graph (DAG) depicting the class hierarchy in the system. SADES schema DAG uses *Version derivation graphs* [LOO 92]. Each node in the DAG is a *class version derivation graph*. Each node in the class version derivation graph (CVDG) has the following structure:

— Reference(s) to predecessor(s)
— Reference(s) to successor(s)
— Reference to the versioned class object
— Descriptive information about the class version such as creation time, creator's identification, etc.
— Reference(s) to super-class(es) version(s)
— Reference(s) to sub-class(es) version(s)
— A set of reference(s) to *object version derivation graph(s)*

Each node of a CVDG keeps a set of reference(s) to some object version derivation graph(s) (OVDG). An OVDG is similar to a CVDG and keeps information about various versions of an instance rather than a class. Each OVDG node has the following structure [LOO 92]:

— Reference(s) to predecessor(s)
— Reference(s) to successor(s)

— Reference to the versioned instance
— Descriptive information about the instance version such as creation time, creator's identification, etc.

Since an OVDG is generated for each instance associated with a class version, a set of OVDGs results when a class version has more than one instance associated with it. As a result a CVDG node keeps a set of references to all these OVDGs. Figure 1 shows the structure of a CVDG node with references to two OVDGs.

The dynamic relationships approach we presented in [RAS 99b] has been employed to implement the SADES conceptual schema. The relationships approach can be employed to extend any existing object database management system and hence has been employed to extend Jasmine with dynamic relationships (see [RAS 99b]). SADES has been layered on top of this Jasmine extension. It is worth mentioning that dynamic relationships can be incorporated into existing Jasmine applications without binding these applications to SADES.

Since the various meta-objects (classes, etc.) that form the SADES schema are interconnected through dynamic relationships dynamic schema changes can be made by dynamically modifying these relationships. Examples are the *derives-from*/*inherited-by* relationships between classes or the *defines/defined-in* relationships between classes and their members. Relationships among meta-objects and objects are used to propagate schema changes to the affected objects. SADES allows a Java programmer to modify the above relationships transparently without concerning him/herself with the underlying schema architecture.

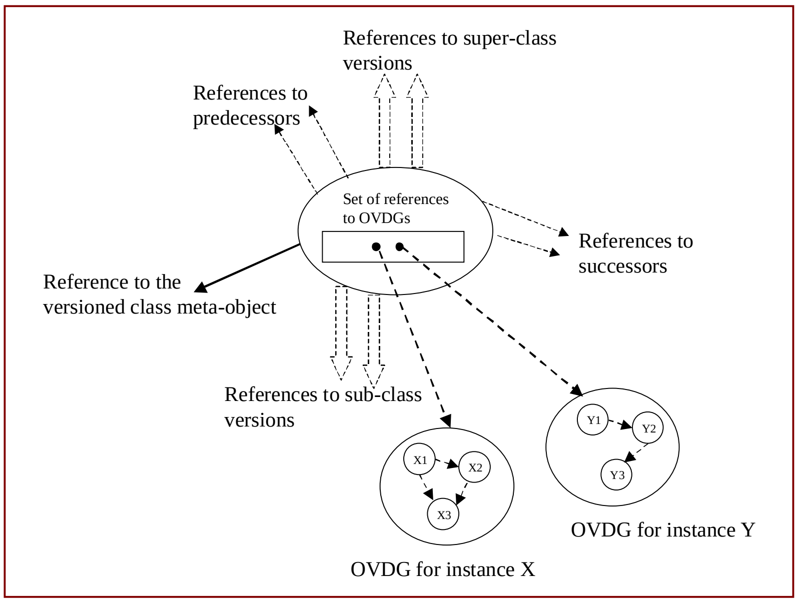
References to super-class versions
References to predecessors
Set of references to OVDGs
References to successors
Reference to the versioned class meta-object
References to sub-class versions
X1
X2
X3
Y1
Y2
Y3
OVDG for instance Y
OVDG for instance X

Figure 1. *A CVDG node with references to two OVDGs*

5.1. *A Layered, Three Tier Architecture*

Transparent access is provided by using the three-tier architecture shown in fig. 2. The SADES server is an RMI server which interacts with the schema model implementation in the dynamic relationships layer on top of Jasmine to obtain both traditional database functionality and evolution functionality. Traditional database functionality is provided by delegating calls to the Jasmine server while evolution functionality is offered by the evolution primitives implemented in the dynamic relationships layer. RMI clients can invoke remote methods implementing the evolution primitives listed in section 3. This, however, would require the programmer to have knowledge about the SADES schema architecture. Furthermore, using RMI would pose a learning curve to programmers not familiar with the technology. Also, the programmer would have to interact with an implementation level object-oriented model and not a higher level conceptual model similar to Java or UML [BOO 97] [QUA 98] object models. As a result the access to evolution functionality would not be transparent.

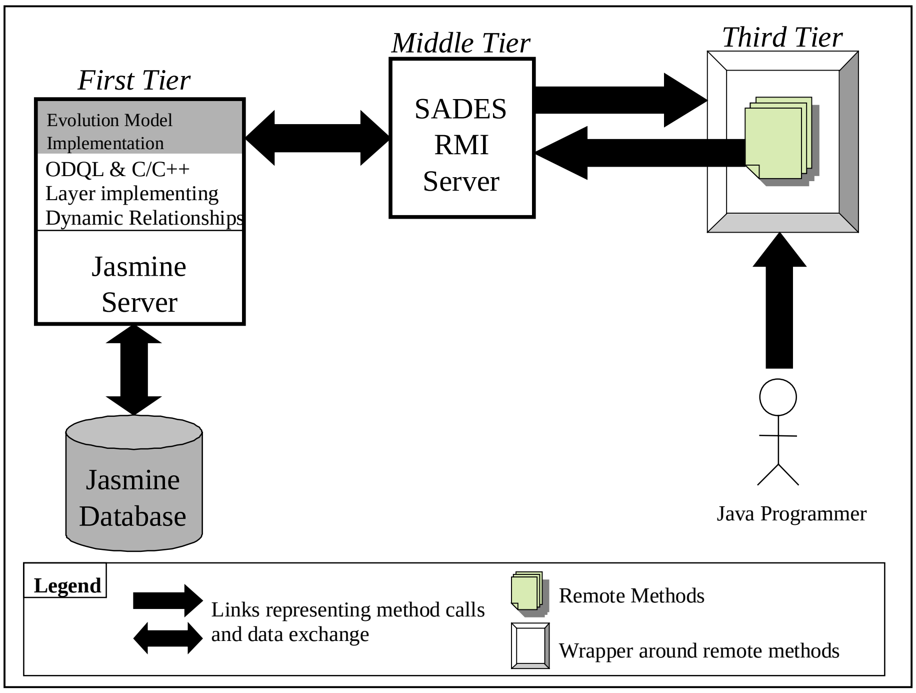
Middle Tier
Third Tier
First Tier
Evolution Model Implementation
ODQL & C/C++ Layer implementing Dynamic Relationships
Jasmine Server
SADES RMI Server
Jasmine Database
Java Programmer
Legend
Links representing method calls and data exchange
Remote Methods
Wrapper around remote methods

Figure 2. *The three-tier architecture used by SADES*

As shown in figure 2 SADES clients address these problems by using wrapper classes instead of invoking remote methods directly. Constructors and member functions in wrapper classes provide wrappers around RMI calls allowing the programmer to interact with the system through a higher level object-oriented model. Instantiation of a wrapper class or invoking a member function on a wrapper class instance results in the call being dispatched to the SADES server transparently of the programmer. The server then performs the required action (interacting with the Jasmine server if necessary) and returns a reference to an object of a wrapper class type. When the client uses the returned reference the wrapped object is fetched from the SADES server. It can then be manipulated in the same way as described above resulting in further RMI calls being dispatched to the SADES server transparently of the programmer.

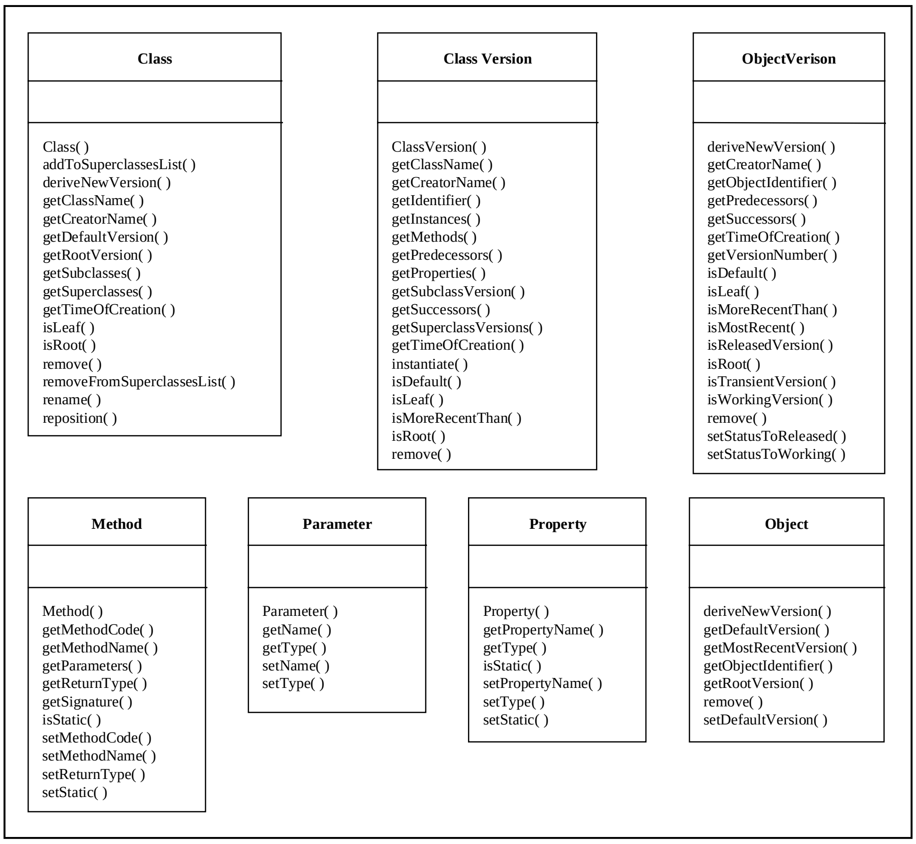

Figure 3. *Wrapper Classes*

5.2. *The Wrapper Classes*

Some of the wrapper classes and signatures for some of their key member functions are shown in fig. 3. Consider, for example, the creation of a new class. The programmer can create a new class by invoking the constructor for the Java class *Class*. Two of the constructor's parameters are *properties* and *methods*. These are arrays of wrapper types *Property* and *Method* and are used to specify the properties and methods for the root version of the new class. The *superClasses* parameter is used to position the new class in the DAG. When the *Class* constructor is invoked the client makes a transparent RMI call to the SADES server. The server then creates the new class at the appropriate position in the DAG and returns a reference to an object of the wrapper type *Class*. Methods can then be invoked on this wrapped object.

In a similar fashion methods of the wrapper class *ClassVersion* can be used to add new properties and methods to an existing class version (resulting in creation of a new class version). Properties and methods can be modified by using member functions of wrapper types *Property* and *Method*[7] respectively.

The layered, three tier architecture and use of wrapper classes provides transparent access to dynamic evolution functionality from Java within the constraints of the language type system. The need to use a read-write reflective mechanism or modify the Java type system is avoided as calls to the database are encapsulated in wrapper class methods and transparently dispatched. The wrapper classes are simple Java classes whose instances represent the meta-objects in the databases. Any structural or behavioural modifications, therefore, only affect the meta-objects represented by wrapper class instances and not the wrapper classes. As a result rules of the Java type system are respected and at the same time transparent access to dynamic evolution functionality is provided.

The transparency achieved using the proposed approach is in direct contrast with the use of ODQL as a declarative change specification language in the Jasmine Java bindings. The use of ODQL as a change specification language results in an impedance mismatch with Java and also poses an intellectual barrier to the programmer who needs to learn a separate language for change specification. Mapping of Java types to ODQL types and vice versa also becomes the programmers responsibility. These problems are effectively addressed by SADES as demonstrated by the following example. The example shows the sequence of operations for changing the superclass of *Student* from *Root* to *Person* using the SADES transparent evolution API:

```
Class studentClass = Class.find("Student");
Class[] superClasses = new Class[1];
superClasses[0] = Class.find("Person");
studentClass.reposition(superClasses, true);
```

It should be noted that the first three lines of code are redundant as references to both *Student* and *Person* classes would have been obtained earlier in the program realising the evolution scenario. The required change is implemented by simply invoking the reposition method for the *Student* class meta-object. The first argument determines its new position in the hierarchy graph while the second indicates that it is to be placed as a leaf class in the hierarchy. The impedance mismatch problem does not exist as the same language is used for programming and change

[7] modifying properties and methods also results in creation of a new class version

specification. Mapping between instances of wrapper types and the corresponding meta-objects in the database is transparently maintained.

6. Summary and Conclusions

We have presented our database evolution system SADES which provides Java programmers transparent access to the evolution functionality of an object-oriented database. The ODMG standard mandates transparent access to traditional object database functionality from Java and all front-line commercially available object database management systems offer Java bindings for the purpose. Therefore, there is a need to provide Java programmers transparent access to advanced object database functionality especially evolution since it is the very characteristic of complex applications which object databases inherently support.

We have layered SADES on top of the commercially available object database management system Jasmine. Traditional database functionality is provided by Jasmine while evolution functionality has been built into the SADES layer. We have shown that although Jasmine provides reasonable evolution features these are not complete. In addition, transparent access to evolution functionality from Java is limited.

SADES provides Java programmers transparent access to all the evolution primitives. The dynamic evolution features operate within the constraints of the Java type system. Programmers interact with a higher level object-oriented model and do not need to be concerned with details of the underlying schema model. At present transparent support for object versioning has been incorporated into the system. A visualisation tool has also been implemented to provide visual access to the various evolution primitives. Our future work will concentrate on incorporating learning and support for evolution of the integrated knowledge-base.

7. References

[AMIE94] AMIEL E., BELLOSTA M.-J., DUJARDIN E., SIMON E., "Supporting Exceptions to Behavioral Schema Consistency to Ease Schema Evolution in OODBMS", *Proc. of 20th VLDB Conf.*, Morgan Kaufmann 1994, pp. 108-119

[ATK 96] ATKINSON M. P., DAYNES L., JORDAN M. J., PRINTEZIS T., SPENCE S., "An Orthogonally Persistent Java", *ACM SIGMOD Record*, Vol. 25, No. 4, Dec. 1996

[BAN 87] BANERJEE J. *et al.*, "Data Model Issues for Object-Oriented Applications", *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, Jan. 1987, pp. 3-26

[BOO 97] BOOCH G., JACOBSON I., RUMBAUGH J., "The Unified Modelling Language Documentation Set", *Version 1.1*, Rational Software Corp., c1997

[CAT 97] CATTELL R. G. G., *et al.*, "The Object Database Standard: ODMG 2.0", Morgan Kaufmann, c1997

[CA 96] "The Jasmine Documentation", Computer Associates International, Inc., Fujitsu Limited, c1996-98

[KIM 88] KIM W., CHOU H., "Versions of Schema for Object-Oriented Databases", *Proc. of 14th VLDB Conf.*, Aug,/Sept. 1988, pp. 148-159

[KIR 96] KIRBY G. N. C., CONNOR R. C. H., MORRISON R., STEMPLE D., " Using Reflection to Support Type-Safe Evolution in Persistent Systems", *University of St Andrews, UK*, Technical Report No. CS/96/10, 1996

[LAU 97] LAUTEMANN S. E., "Schema Versions in Object-Oriented Database Systems", *Proceedings of the 5th International Conference on Database Systems for Advanced Applications (DASFAA)*, April 1997, pp. 323-332

[LOO 92] LOOMIS M. E. S., "Object Versioning", *Journal of Object Oriented Programming*, Jan. 1992, pp. 40-43

[MON 93] MONK S., SOMMERVILLE I., "Schema Evolution in OODBs Using Class Versioning", *SIGMOD Record*, Vol. 22, No. 3, Sept. 1993, pp. 16-22

[QUA 98] QUATRANI T., "Visual Modelling with Rational Rose and UML", Addison Wesley, c1998

[RA 97] RA Y.-G., RUNDENSTEINER E. A., "A Transparent Schema-Evolution System Based on Object-Oriented View Technology", *IEEE Transactions on Knowledge and Data Engineering*, Vol.9, No.4, July/Aug.1997, pp. 600-624

[RAS 98a] RASHID A., SAWYER P., "Facilitating Virtual Representation of CAD Data through a Learning Based Approach to Conceptual Database Evolution Employing Direct Instance Sharing", *Proceedings of the 9th International Conference on Database and Expert Systems Applications*, Aug. 1998, LNCS 1460, pp. 384-393

[RAS 98b] RASHID A., "SADES - a Semi-Autonomous Database Evolution System", *Proceedings of the 8th International Workshop of Doctoral Students in Object Oriented Systems*, Jul. 1998, ECOOP '98 Workshop Reader, LNCS 1543

[RAS 99a] RASHID A., SAWYER P., "Evaluation for Evolution: How Well Commercial Systems Do", *Proc. of 1st Workshop on OODBs held in conjunction with 13th European Conference on Object Oriented Programming*, June 1999, Lisbon, Portugal, pp. 13-24

[RAS 99b] RASHID A., SAWYER P., "Dynamic Relationships in Object Oriented Databases: A Uniform Approach", *Proceedings of the 10th International Conference on Database and Expert Systems Applications*, Aug.-Sept. 1999, LNCS 1677, pp. 26-35

[RAS 00a] RASHID A., SAWYER P., "Toward 'Database Evolution': a Taxonomy for Object Oriented Databases", *Cooperative Systems Engineering Group, Computing Department, Lancaster University*, Technical Report No: CSEG/05/00

[SKA 86] SKARRA A. H., ZDONIK S. B., "The Management of Changing Types in an Object-Oriented Database", *Proceedings of the 1st OOPSLA Conference*, Sept. 1986, pp.483-495

[SJO 93] SJOBERG D., "Quantifying Schema Evolution", *Information and Software Technology*, Vol. 35, No. 1, pp. 35-44, Jan. 1993