

Evaluation for Evolution: How Well Commercial Systems Do

Awais Rashid¹, Peter Sawyer¹

¹Cooperative Systems Engineering Group, Computing Department, Lancaster University, Lancaster LA1 4YR, UK
{marash, sawyer} @comp.lancs.ac.uk

Abstract. Like any other database application object database applications are subject to evolution. Evolution is, however, critical in object databases because it is the very characteristic of complex applications such as CAD, etc. for which they provide inherent support. This paper discusses the evolution facilities offered by some of the existing systems. We first provide an overview of the ODMG 2.0 standard from an evolution viewpoint. We then describe our extension to the database evolution features specified by the ODMG 2.0 standard. Using this extension as a basis we form evaluation criteria, which we employ to assess the various evolution facilities offered by four commercially available object database management systems: POET, Versant, O2 and Jasmine.

1. Introduction

Object oriented databases come with a promise to address integrated and advanced applications such as computer-aided design (CAD), computer-aided manufacturing (CAM), computer-aided software engineering (CASE) and office automation systems, etc. The conceptual structure of a database application is not as constant as it may appear to be and can vary to a large extent [24]. 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 artifacts modeled in the database. Therefore, like any other database application object database applications are subject to evolution. An important requirement in terms of evolution is to keep track of the changes in order to provide backward compatibility with applications accessing the database prior to the changes and in case the changes need to be reverted. Also, over the time period when the conceptual structure of the database is stable, applications such as computer aided design (CAD), etc. still require keeping track of evolution of the data residing in the database. Evolution, therefore, is critical in object oriented databases since it is the very 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 modeled 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.

Therefore, 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 evolve, through various versions, both the objects and the class definitions.

Historically, the database community has employed three fundamental techniques for modifying the conceptual structure of a database, namely:

- schema evolution [1, 6], where the database has one logical schema to which class definition and class hierarchy modifications are applied
- schema versioning [5, 8, 9, 10, 15, 16, 17, 20], which allows several versions of one logical schema to be created and manipulated independently of each other
- class versioning [12, 13, 23], which keeps different versions of each type and binds instances to a specific version of the type

In contrast to the above three approaches [21, 22] superimpose 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 [21, 22] approach as a model for database evolution, this paper provides a survey of dynamic database evolution facilities offered by four commercially available object database management systems:

- POET¹
- Versant²

¹ POET Software, <http://www.poet.com/>

- O2³
- Jasmine⁴

Only a subset of the front-line commercially available object database management systems has been surveyed due to time and financial constraints. Other front-line commercially available systems include Object Store⁵, Objectivity⁶ and Gemstone⁷. The free availability of POET, Versant and Jasmine from their respective vendors for evaluation purposes led to their choice, while O2 has been licensed from Ardent Software by the Computing Department, Lancaster University for research purposes. We first present an overview of the ODMG 2.0 standard [3] from a database evolution perspective. We then describe our extension to the database evolution features specified by the ODMG 2.0 standard. Using this extension as a basis we form evaluation criteria, which we employ to assess the various evolution and versioning facilities offered by the four selected commercial ODBMSs. The discussion is, however, mainly limited to the C++ and Java bindings of the respective ODBMS. Where relevant other language bindings have been discussed. The last section summarises and concludes the paper showing how closely each of the four ODBMS products matches our evaluation criteria.

2. Evolution and the ODMG Standard

ODMG, the Object Data Management Group formed in 1991, is a non-profit consortium of vendors and interested parties who collaborate to develop and promote standards for object storage. Since its emergence in 1993 [2] the ODMG standard has become the defacto standard for object databases. Over the years there have been increasing efforts by ODBMS vendors to market products conformant to the standard. At present there exist several *ODMG compliant* ODBMSs in the market while there are others which implement part of the standard.

2.1. The Standard

The standard aims at providing application portability across various ODBMS products. For this purpose the standard defines an object model to be supported by compliant ODBMSs. The standard also describes two object specification languages, namely the Object Definition Language (ODL) and the Object Interchange Format (OIF). It also provides declarative, non-procedural access to the database objects through the Object Query Language (OQL). It mandates standard ODBMS bindings for C++, Smalltalk and Java allowing portability of code that manipulates the persistent objects. The standard, however, is still evolving and has not fully matured. Most of the efforts have been concentrated on providing traditional database functionality and ensuring portability across vendor products. It does not yet cover features such as dynamic schema evolution, object versioning and workgroup support. The C++ binding, however, specifies a *read-only* schema access API as a first step towards providing dynamic schema evolution. The following two sections provide an overview of the ODMG C++ and Java bindings from a database evolution viewpoint.

2.2. ODMG C++ Binding

The ODMG C++ binding provides the necessary constructs to implement the concepts defined in the ODMG object model. Besides specifying the logical characteristics of objects and the operations used to manipulate them the C++ binding also provides a schema access interface for an ODMG database. The present specification of the schema access interface provides only *read-only* access to the schema hence disallowing dynamic creation and modification of classes. The ODMG, however, plans to extend the specification to *read/write* so that dynamic modifications to the database schema are possible [3].

The *read-only* schema access API can be used to obtain comprehensive information about the various meta-objects in the database. It also provides facilities to iterate over the various relationships these participate in with other meta-objects. In the ODMG schema model classes, types, defining scopes, class properties, class methods, etc. together form the meta-object space. The relationships between meta-objects therefore include not only the *derives from/inherits to* relationship between a sub-class and its super-class but also the *defined in/defines* relationship between a meta-object and its defining scope. The schema access API thus provides all the constructs needed to traverse and interrogate the conceptual schema of an ODMG compliant database.

The ODMG standard does not specify any object versioning strategies or support for group work and the C++ binding does not provide any facilities for these.

² Versant Object Technology, <http://www.versant.com/>

³ Ardent Software, <http://www.ardentsoftware.com/>

⁴ Computer Associates International, Inc. & Fujitsu Limited, <http://www.cai.com/>, <http://www.fujitsu.com>

⁵ Object Design Inc., <http://www.objectdesign.com/>

⁶ Objectivity Inc., <http://www.objectivity.com/>

⁷ Gemstone Systems Inc., <http://www.gemstone.com/>

2.3. ODMG Java Binding

The Java programming language has become highly popular in the industry over the recent years and is, likewise, finding increasing interest among the object database community.

One of the major increments in ODMG 2.0 over ODMG-93 is the inclusion of a Java binding. The binding provides the Java programmer with transparent access to the database. However, unlike the C++ binding it does not support schema access at present. The possibilities to achieve dynamic schema evolution through the existing Java binding are further precluded by the fact that the mechanism adopted to make Java objects persistent is highly static. The ODMG Java binding requires classes whose instances might be stored in the database to be explicitly marked *persistence capable*. This is achieved either at the pre-compilation stage through a *preprocessor* which modifies the Java source code before it is compiled by the Java compiler, or at the post-compilation stage through a *postprocessor* which modifies the byte code produced by the Java compiler. Although they help to provide seamless access to the database, both mechanisms are static in nature since they bar the system from running uninterrupted when new classes are introduced. The new classes need to be *pre/post processed* in order to be augmented with the code necessary to make their instances persistent. The problem is made worse if the classes being added form non-leaf nodes in the class hierarchy as *pre/post processing* of both the super-classes and sub-classes is required.

3. Extending the ODMG Database Evolution Features

In this section we present our extension to the database evolution features specified by the ODMG 2.0 standard. We categorise the database evolution features desirable of an ODBMS into:

- ODMG implicit features
- Extended features

Features that provide read-only access to the conceptual structure of a database are characterised as ODMG implicit. These, therefore, offer functionality similar to that provided by the ODMG C++ Schema Access API i.e. allowing traversal of the meta-object hierarchy and providing access to the structure of the meta-objects in the system.

Extended features allow dynamic modifications to the conceptual structure of the database; functionality not yet offered by the ODMG C++ Schema Access API. The extended features are further subdivided into three categories:

1. Class hierarchy evolution features
2. Class structure evolution features
3. Workgroup support features

The class hierarchy evolution features allow dynamic addition and deletion of classes that form both leaf and non-leaf nodes in the class hierarchy graph in the system. It should be noted that in our extension the deletion of a class that forms a non-leaf node in the class hierarchy graph does not delete all its sub-classes unless so desired. Instead, inheritance relationships are updated so that the sub-classes of the deleted class become sub-classes of its immediate super-class(es).

The class structure evolution features allow dynamic modifications to the existing classes in the system. These offer functionality to dynamically add, drop and modify both the class attributes and methods. Support is also provided for both *linear* and *branch* [7] class versioning.

The workgroup support features provide *linear* and *branch* [7] object versioning facilities. They extend the conventional transaction model and read/write locking mechanism to long transactions and advanced persistent locks respectively in order to allow check-in/check-out of objects from the shared database to private workspaces. Advanced persistent locks have properties additional to read/write in order to resolve locking conflicts between long transactions started by different users.

4. Evolution in Commercial ODBMSs

In this section we present a discussion of the database evolution facilities offered by each of the four selected object database management systems. We first form evaluation criteria based on our extension to the database evolution features specified by the ODMG 2.0 standard. We then use these as the basis to assess each of the selected ODBMS products.

Our evaluation criteria require an ideal ODBMS to possess the following three sets of features:

1. Class hierarchy access and evolution features

The ODBMS should possess the ODMG implicit features for traversing the class hierarchy in the system. It should also possess the extended features allowing dynamic addition and deletion of classes that form both leaf and non-leaf nodes in the class hierarchy graph.

2. Class structure access and evolution features

The ODBMS should offer the ODMG implicit functionality to access the structure of existing classes in the system. It should also provide facilities for dynamically modifying the class structure allowing addition, deletion and modification of both class attributes and methods. It should also support both linear and branch class versioning.

3. Workgroup support features

The ODBMS should facilitate both linear and branch object versioning. In addition, it should support long transactions and advanced persistent locking mechanisms for check-in/check-out of objects from the shared database to private workspaces.

We now discuss the database evolution facilities offered by the four ODBMSs: POET, Versant, O2 and Jasmine respectively. Each system is assessed on the basis of the evaluation criteria described above. The discussion concentrates on the database evolution facilities offered through the C++ bindings. Functionality offered through the Java bindings has also been summarised. Where relevant, other language bindings have been discussed.

4.1. POET

The POET object database management system from POET Software is ODMG compliant. POET offers C++ and Java bindings which are highly conformant to ODMG 2.0. However, only a subset of OQL is supported. The experiences described in this section are based on POET 5.0 [19] running on Windows NT 4.0.

4.1.1 POET C++ Binding

POET offers two C++ bindings each of which can be used either stand alone or, within certain limitations, in combination with the other. The POET ODMG C++ binding is tailored to meet the requirements specified in ODMG 2.0. However, it does not provide the schema access API that is part of the standard ODMG C++ binding. The other binding is POET specific and in addition to providing traditional database functionality also provides advanced features such as dynamic creation and modification of classes and workgroup support. The workgroup support is however limited to providing check-in/check-out of objects from the shared database to private workspaces through long transactions and persistent locks. The binding does not support object versioning for objects being checked-in or checked-out. The persistent locks are limited to the conventional *read-only* or *read-write* characteristics thus barring other users from modifying the object once it has been checked out. For databases sharing the same schema the binding allows objects with cross-database references to be checked out into private workspaces but such cross-database references are not managed at the DBMS level and hence must be dealt with at the application level.

For dynamic schema access and modification the binding offers constructs to:

- traverse the class hierarchy
- retrieve information about a particular class such as the class name, class identifier, references to other meta-objects that comprise the class members and which classes the class is derived from or inherits to
- retrieve for each class member a description of its:
 - storage class (*static*, *extern*, *automatic*, *register* or *inline*)
 - qualifiers (*const* or *volatile*)
 - access type (*private*, *protected* or *public*)
 - access methods i.e. direct access (using a reference &) or indirect access (using a pointer *)
- dynamically create and drop classes
- dynamically modify the class structure

While creating a new class it is possible to specify its super-classes and whether the class would be *instantiable* or *abstract*. It is also possible to specify whether the instances of the new class being created would be *transient* or *persistent*. Deleting classes is a bit inconvenient as POET does not automatically update relationships between various meta-objects upon deletion of one. It is, therefore, not possible to delete a class if some other class has a member whose type is the class being deleted. One needs to drop either the data member or the whole class in order to delete the required class. Dropping a non-leaf class drops all of its sub-classes. This is undesirable behaviour in many cases.

One can drop and add data members of both basic and user defined types to a class. The lifetime of the class member (*transient* or *persistent*), its storage class, qualifiers, access type and access method can also be specified. However, there is no facility to specify *friends*; constructs that are not class members but have the privilege to access the private parts of a class. The binding allows changing the type of a class member without dropping and recreating. Renaming a class member is also possible but since POET maintains relationships between various parts

of a class description through names of the components this is viewed as equivalent to dropping a class member and adding a new one.

When modifying classes already existing in the database schema, POET offers class versioning facilities. The user does not have direct control over whether a new class version is created upon modification. However, it is possible to achieve this in an indirect manner by opting either to modify the class definition that was retrieved or its *clone*, an exact copy of the retrieved class definition. In the latter case, a new class version is generated by default. Creation of a new class version upon modification of the original class definition is desirable as existing instances of the class would become inaccessible otherwise. Older class versions are accessible by traversing the *version derivation tree* starting at the latest version. Note that the version history in POET is a tree and not a graph since POET provides only *linear versioning* [7] for classes. Each class version being created has only one predecessor resulting in a version derivation tree. Instances created using older class versions remain accessible through their respective class versions. These are converted to the latest class version on-the-fly if accessed using the latest version. One can, however, choose to read the instances using the latest class version but with the modifications not written to the database and vice versa. It is also possible to explicitly update all the instances of the versioned class to the latest version. This might be desirable upon deletion of an older class version; with the C++ binding it is possible to delete older class versions. The limitations of class versioning in POET are:

- *Branch versioning* [7] for classes is not supported
- Dynamic modification and versioning of classes work best for leaf nodes in the class hierarchy. Versioning or modification of non-leaf classes results in instances of their sub-classes becoming invalid

4.1.2 POET Java Binding

POET offers a Java binding highly conformant to ODMG 2.0. In addition to providing the standard ODMG features, POET's Java binding also provides additional functionality such as maintaining extents⁸, providing user login information and read-only access to object identifiers. These are features not yet supported by the ODMG standard. The high conformance to the ODMG standard is, however, a drawback in terms of dynamic schema evolution as POET employs a preprocessor to mark persistence capable classes. As discussed earlier this model for achieving persistence is highly static and does not leave room for dynamically modifying the database schema. POET's Java binding does not yet provide any workgroup support and object versioning facilities.

4.2. Versant

Versant, from Versant Object Technology, is also one of the ODMG compliant databases on the market. Versant, however, does not support OQL. Instead, it provides VQL, Versant Query Language, which is also quite close to SQL. The experiences presented in this section are based on Versant Release 5.0.7 [26] running on Windows NT 4.0.

4.2.1 Versant C++ Binding

Like POET, Versant provides two C++ bindings. In addition to its own DBMS binding for C++, Versant offers a C++ binding compliant with ODMG Release 1.2. The Versant ODMG C++ binding, therefore, does not provide the schema access API, which is part of the standard C++ binding in ODMG Release 2.0. Unless otherwise stated, from this point onwards, any references to the Versant C++ binding refer to the non-standard vendor specific binding and not the ODMG binding.

The Versant C++ binding provides facilities for dynamic modification of the database schema. For this purpose the binding provides constructs to create leaf classes allowing the creator to specify the superclass(es) of the new class being created. The Versant C++ binding also provides facilities to drop leaf classes and rename both leaf and non-leaf classes. It is also possible to modify inheritance relationships by dropping a class from the super-class(es) list of another class. This, however, does not remove the dropped super-class from the actual database schema. In addition, adding, dropping and renaming of attributes is possible for both leaf and non-leaf classes. Traversing the class hierarchy is also possible. It should be noted that classes can be created and modified dynamically as long as they are simple classes and do not use virtual functions or virtual inheritance [25].

The C++ binding also provides workgroup support through object check-in/check-out, long transactions and persistent locks. Locking conflicts with persistent locks are dealt with by providing additional properties to locks in addition to being read/write. A persistent lock can be a *soft lock* which means it can be broken or a *hard lock* that cannot be broken by long transactions started by other users. Also, upon requesting a persistent lock it is possible to specify *waiting* or *reservation* requests in case another user has checked out the object. Versant provides facilities to notify a user of reservation requests from other users; the availability of an object that was reserved; or the breaking of a soft lock on an object checked out by the user. The binding also provides facilities for object versioning and

⁸ The extent of a type is the set of all its instances within a particular database

maintenance of *version derivation graphs* [11] i.e. *branch versioning* [7] for objects is supported. Versions of *versionable* objects can be checked in and checked out of the shared database to private workspaces. In addition to keeping track of data evolution this eliminates the need for holding persistent locks on objects as a new version is automatically created upon checking out or checking in versioned objects. Each versioned object has a *default version* [11] which can be changed. It is also possible to assign a status to a version i.e. it can be a *transient version*, *working version* or a *released version* [11]. Versant maintains information about the most recent transient, working and released versions of an object and also the most recent version of the object. This information is also accessible through the C++ binding.

4.2.2 Versant Java Binding

The Versant Java binding offers three main Java packages for accessing the database namely the *transparent package*, the *ODMG package* and the *fundamental package*. As its name suggests the transparent package provides seamless access to the database. The same is true of the ODMG package. However, it is tailored to meet the requirements specified in the ODMG 2.0 Java binding and therefore suffers from the same drawbacks, in terms of database evolution, as the ODMG Java binding. Unlike its other two counterparts, the fundamental package provides direct access to the database. Like the Versant C++ binding, it offers constructs to dynamically create and drop leaf classes and rename both leaf and non-leaf classes. Facilities are also available to traverse the class hierarchy and add, drop and rename attributes in both leaf and non-leaf classes. The Versant Java binding, however, does not yet provide any facilities for object versioning or workgroup support.

It should be noted that both Versant C++ and Java bindings allow dynamic synchronization of class definitions in different databases. It is also worth noting that Versant takes a *deferred update* approach rather than an *immediate update* approach to schema evolution i.e. instances of modified classes are not immediately updated to reflect changes in the class structure. Rather, changes are applied on-the-fly as instances are accessed. This helps avoid paralyzing the database by retrieving and updating all the instances of a class each time the class definition or its superclass(es) definition(s) is modified. However, it leads to an inconsistency between the data and the meta-data. Versant addresses the problem through maintaining version histories internally. Versant, however, does not currently provide any facilities for prohibiting individual users from modifying the database schema. This can lead to unwanted modifications of the database schema on the part of some users affecting all the users. Neither does it provide any facilities for class versioning or schema versioning in order to provide different views of the schema to different users. The version history internally maintained during schema evolution is invisible to the user barring one from keeping track of schema modifications in the system.

4.3. O2

The O2 object database management system from Ardent Software is also one of the commercially available ODMG compliant systems. Besides providing a C++ binding highly conformant to ODMG Release 1.2 and support for OQL, O2 offers programming language interfaces for C, Java and O2C. O2C is a 4th Generation Language and a superset of C. O2 offers a wide range of dynamic schema manipulation and object versioning facilities fully accessible through O2C. We, however, limit our discussion to accessing these facilities from the C/C++ and Java APIs only. The experiences presented in this section are based on the O2 System Release 5.0 [14] running on Windows NT 4.0.

4.3.1 O2 C++ Binding

O2 offers a C++ binding highly conformant to ODMG Release 1.2. In addition to the standard C++ class libraries, it offers a C++ class library providing object versioning facilities. It, however, does not offer any object check-in/check-out facilities for workgroup support. As it is conformant to release 1.2, the O2 C++ binding does not include the schema access API that is part of the standard C++ binding in ODMG 2.0. O2, however, provides a range of dynamic schema modification facilities through the O2Engine API. This is a C API, which provides a C/C++ programmer with direct access to the O2 database engine functionality. We first discuss the object versioning facilities offered by the C++ binding and then examine the various dynamic schema manipulation facilities available through the O2Engine API.

O2 allows a C++ programmer to create and manipulate versions of objects. All objects that need to be versioned must participate in a *configuration*, a collection of objects, which forms the unit of versioning. There is no restriction as to which types of objects can be versioned. Versioned objects can reference non-versioned objects and vice versa. Therefore, if all the objects being referred to by a versioned object also need to be versioned they must explicitly be put into the configuration their referring object participates in. Cross database references are also allowed and are managed transparently by O2. Branch versioning is supported hence the resulting version derivation hierarchy is a graph. More than one version derivation graph can exist in the system since multiple configurations can co-exist. An object, however, can only belong to one configuration. Navigating through the version derivation graph is easy since

the class library provides constructs to find the parent versions of a particular version or the versions derived from it. It is also possible to directly access the *root version* in the derivation graph.

Labels can be assigned to versions in the derivation graph. Versions can later be accessed through their respective labels. Labels can be modified in due course. Access modes can be specified for a particular version which can be *read-only* or *read-write*. It is possible to specify a *default version* and a *current version* for the particular version derivation graph. New versions can be derived from existing versions and existing versions deleted. Deletion of some particular versions, such as the root version, default version and a read-only version, is not allowed. The current version can be deleted but the execution context has to be updated afterwards. The system also allows the merging of two versions in the same derivation graph. However, the operation is abstract in nature as the merging is only at the derivation graph level. The actual contents of the objects that form part of the two versions are not merged. This has to be carried out by the application. The system, however, aids the process by providing constructs to find the difference between the contents of two versions belonging to the same graph. It should also be noted that a versioned object (which is part of a configuration) can be removed from the configuration any time and hence *de-versioned* and used as other non-versioned objects in the database.

It is possible to dynamically modify the database schema from a C++ application using the O2Engine API which provides direct control of the schema, classes, objects and transactions. Using the API it is possible to navigate through the class hierarchy and structure of the classes that form the hierarchy. The API functions allow the dynamic creation of new classes and the deletion of existing classes in the schema. However, only classes that form leaf nodes in the class hierarchy graph can be added or deleted. Note that it is possible to delete a class with existing references to it. The system replaces the deleted class with an *undefined* class of the same name. Such undefined classes can be detected through a validation process and can be removed or redefined. Dynamically creating a new class requires the specification of its name, inheritance links, member function(s) signature and, if desired, renaming inherited class members. Since class hierarchy in O2 is single-rooted, if no super-class(es) is specified for the new class being created, it inherits from the system class *Object* by default. O2 validates the class creation process by checking if all the super-classes specified exist and whether the structure of the new class is compatible with that of its super-classes.

In case naming conflicts occur and no renaming has been carried out explicitly the system implicitly performs the renaming. Both attributes and methods can also be redefined in a sub-class as long as sub-typing rules are adhered to. New members, both attributes and methods, can be added and existing ones removed or modified with the change propagated to all the sub-classes. Defining a new attribute requires specification of its name and type while a new method can be defined specifying its name, return type and names, types and the number of parameters. Note that if not explicitly provided each method takes an implicit parameter referring to the object on which it is invoked. Class method code is compiled separately and through the O2Engine API can be stored in the working schema. Facilities are also available for dynamic linking and loading of class methods. A point worth noting is that O2 identifies class members, both attributes and methods, through a unique identifier. This provides an unambiguous way of recognising the class member since it has the same identifier in all the classes where it is found either by inheritance or definition and whether it has been redefined or not. The API also provides functions to help maintain consistency between objects in the database and the class definitions during evolution. Facilities to make definitions of one or more classes in the working schema available to other schema are also provided. No class versioning facilities are offered however.

4.3.2 O2 Java Binding

O2 offers a Java binding to access both relational databases and the O2 object database through Java. The binding is not conformant to the ODMG 2.0 Java binding. It is, however, greatly influenced by the ODMG Java binding and in order to provide transparent access to an O2 database it uses a class file postprocessor which augments the Java byte code with methods required to manage persistence. The O2 Java binding, therefore, suffers from the same drawbacks as its ODMG peer in terms of database evolution. The binding does offer advanced functionality such as maintenance of extents and user management and database access control management. However, it does not yet offer any dynamic schema manipulation and versioning facilities. No support for workgroups is available.

4.4. Jasmine

The Jasmine object database management system is a joint venture of Computer Associates International, Inc. and Fujitsu Limited. Unlike POET Software, Versant Object Technology and Ardent Software, who are all *voting members* of the ODMG, Computer Associates International, Inc. are a *reviewer member*. Jasmine, therefore, does not come with the promise to be conformant with the ODMG standard. However, it 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. One of the Jasmine Java bindings is quite close to the ODMG Java binding and supports OQL.

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. The contents of this section are based on experiences with Jasmine 1.2 [4] on Windows NT 4.0. We first discuss the evolution facilities available through ODQL and then extend our discussion to using these facilities through C++ or 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.

4.4.1 ODQL

ODQL provides facilities to access the database meta-data at run-time. Using the various system-provided operations it is possible to traverse the class hierarchy in each *class family* in the database. Note that Jasmine uses *class families* as namespaces in a way similar to Java *packages*. Besides traversing the *superclass-subclass* relationships among classes, class properties are also accessible. Jasmine stores the signature and source code of class methods as part of the meta-data. ODQL, therefore, offers operations to access both the method signature and the source code. Besides providing access to database meta-objects, ODQL also offers functionality to both dynamically create new meta-objects and modify existing ones.

Using ODQL a new class can be defined dynamically by specifying its name, class family, a short description, super-class(es), properties, member function(s) signature and the maximum size the instances of the class could have. If the specified super-class(es) does not exist at the time the new class is being defined, Jasmine creates a temporary class for the undefined super-class(es). Since Jasmine is *single-rooted*, if no super-class(es) is specified the new class inherits from the system class *Composite* by default. Besides specifying the data type, name, a descriptive comment and whether a property of the new class being created is unique, mandatory and/or has a default value it is also possible to specify if it is an *instance-level* (ordinary member) or a *class-level* (*static*) property.

The new class definition can also include a specification of its member function(s) signature. Only member functions' signatures are specified. The member function implementations for the class are carried out separately from the class definition. ODQL offers facilities to dynamically compile member function implementations for a class and to pass options to the host language compiler if embedded in a host language (C or C++). The inclusion of the signature in the class definition is, therefore, optional as they can be specified at the implementation time. The signature specification includes the name of the method, return type and names and types of its parameters. A default value for a parameter may also be specified. Like properties, methods can also be class-level or instance-level. There is, however, no way of specifying the access type (*private*, *public*, *protected*) to a class member (both properties and functions). It should be noted that the new class definition must be validated before instances of the class can be created. The ODQL validation process checks for problems such as name clashes, undefined classes that are referred to by the new class definition, etc.

Dynamic modifications to the class structure and the class hierarchy are also possible. With ODQL it is possible to add or remove a sub-class to an existing class at any time. However super-classes can be added or removed from a class only if it has not yet been validated. Once a class has been validated it needs to be deleted and redefined if its super-classes need to be changed. Although in such cases existing data can be preserved through careful use of the Jasmine unload and load utilities, object identifiers would change as a result. This is an undesirable behavior since any references to the older objects need to be updated by the application. It should also be noted that deleting a class results in deletion of not only its instances but also all its sub-classes and their instances. This again is a behavior which can be undesirable in many cases. Also, Jasmine does not automatically adjust any relationships from other classes to the deleted class. Such references have to be managed at the application level.

Properties can be added to or dropped from existing classes in the database schema. When a new property is added the change is immediately propagated to all the instances and sub-classes of the class. Jasmine validates the property addition process and disallows any additions that cause name clashes within the class or any of its sub-classes. Deleting a property deletes it from all the instances and sub-classes of the class. Existing class properties can be dynamically modified as well. ODQL allows creation and removal of the uniqueness or mandatory constraints for a property. It is also possible to specify a new default value or change the existing one for the property being modified. More complex changes to a property require creating a new property and deleting the existing one.

Like class properties, new class methods can be added dynamically and existing ones removed. The source code of a class method can also be revised and re-compiled dynamically. Adding a new parameter or dropping an existing one requires creating a new method with the same name and the new signature and deleting the existing one.

4.4.2 Jasmine and C++

Jasmine offers a C API which can be used to access Jasmine databases from C++. The C API provides facilities to execute ODQL statements and commands passed as parameters to functions written in C. The C++ programmer can, therefore, make use of full ODQL functionality including dynamic schema access and modification. It should, however, be noted that there is no correspondence between C++ classes and Jasmine classes (created through ODQL). Any such mappings have to be dealt with at the application level. Although Jasmine provides facilities to map ODQL variables to C variables and vice versa it suffers from the *impedance mismatch problem* since the language used to access the database is different from the one used to manipulate the data. Although ODQL has very rich semantics and functionality any complex computations need to be carried out in C or C++ and mapping ODQL classes to C++ classes and vice versa can be very expensive and cumbersome. The Jasmine Java bindings, however, try to bridge the gap between the programming language and the database language as discussed in the next section.

4.4.3 Jasmine Java Binding

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 with 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. As discussed earlier 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. pJ gets around the problem by providing access to the whole ODQL functionality in a style similar to the C API. 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 similar to the C API in order 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)*⁹ 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 and the C API 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.

5. Summary and Conclusions

We have discussed the ODMG standard and four commercially available object database management systems exploring the various dynamic database evolution facilities available. The ODMG C++ binding includes a schema access API as a first step towards providing dynamic schema manipulation facilities. Commercial object database management systems also realise the importance and need for advanced functionality such as object versioning, workgroup support and dynamic schema manipulation. Tables 1, 2 and 3 below sum up the results of our evaluation of the four systems showing how closely each meets our evaluation criteria.

From table 1 we see that none of the systems allows the dynamic addition and dropping of non-leaf classes. They fall short of meeting our criteria for class hierarchy access and evolution. All four systems offer similar functionality through either the C++ or Java binding. Versant and Jasmine stand out as the same set of features is available through both the bindings.

Table 2 provides a comparison of the class structure access and evolution features offered by the four DBMSs. Again, neither of them fully satisfies our evaluation criterion. Jasmine, however, gets close by providing, through both the C++ and Java bindings, all the desirable functionality except class versioning. O2 follows by offering the same set of features accessible through C++. It is also worth noting that POET is the only system that offers class versioning facilities through its C++ binding.

⁹ <http://java.sun.com/products/jndi/>

The workgroup support features offered by each of the systems have been compared in table 3. Versant is outstanding as the features offered by its C++ binding fully meet our evaluation criterion.

The paper discusses the evolution features offered by a subset of front-line commercially available object database management systems. The remaining three front-line systems (Object Store, Objectivity and Gemstone) not covered in the paper also realise the need for evolution and provide functionality for the purpose. Object Store, for example, offers a meta-object protocol through a set of C++ classes [18] which provide run-time read access to the database schema and facilities for dynamic type creation and modification. The Object Store C++ binding also offers object versioning facilities and workgroup support. Classes created dynamically are, however, not compiler-heterogeneous. Also, object identifiers of affected instances change upon migration to the modified schema since Object Store performs the migration by creating new instances based on the modified class definitions and copying data from old instances into the new ones. We aim at including Object Store in the study since it claims the biggest share of the ODBMS market. This will be followed by evaluation of the five systems (Object Store, POET, Versant, O2 and Jasmine) on the basis of other criteria such as performance, scalability and support for distributed applications. Gemstone and Objectivity will also be included in the evaluation if possible.

Facilities ODBMS		Traversing meta-object hierarchy	Non-leaf classes		Leaf classes	
			Addition	Deletion	Addition	Deletion
POET	C++	√	×	×	√	√
	Java	×	×	×	×	×
Versant	C++	√	×	×	√	√
	Java	√	×	×	√	√
O2	C++	√	×	×	√	√
	Java	×	×	×	×	×
Jasmine	C++	√	×	×	√	√
	Java	√	×	×	√	√

Table 1: Comparison of class hierarchy access and evolution features

Facilities ODBMS		Meta-object structure access	Class versioning		Attributes			Methods		
			Linear	Branch	Addition	Deletion	Modific- ation	Addition	Deletion	Modific- ation
POET	C++	√	√	×	√	√	√	×	×	×
	Java	×	×	×	×	×	×	×	×	×
Versant	C++	√	×	×	√	√	√	×	×	×
	Java	√	×	×	√	√	√	×	×	×
O2	C++	√	×	×	√	√	√	√	√	√
	Java	×	×	×	×	×	×	×	×	×
Jasmine	C++	√	×	×	√	√	√	√	√	√
	Java	√	×	×	√	√	√	√	√	√

Table 2: Comparison of class structure access and evolution features

Facilities ODBMS		Check-in/ Check-out	Long Trans.	Persistent Locks		Object Versioning	
				Read/ Write	Advanced	Linear	Branch
POET	C++	√	√	√	×	×	×
	Java	×	×	×	×	×	×
Versant	C++	√	√	√	√	√	√
	Java	×	×	×	×	×	×
O2	C++	×	×	×	×	√	√
	Java	×	×	×	×	×	×
Jasmine	C++	×	×	×	×	×	×
	Java	×	×	×	×	×	×

Table 3: Comparison of workgroup support features

References

- [1] 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
- [2] Cattell, R. G. G., *et al.*, “The Object Database Standard: ODMG-93 Release 1.2”, *Morgan Kaufmann*, c1995
- [3] Cattell, R. G. G., *et al.*, “The Object Database Standard: ODMG 2.0”, *Morgan Kaufmann*, c1997
- [4] “The Jasmine Documentation”, *Computer Associates International, Inc., Fujitsu Limited*, c1996-98
- [5] Kim, W., Chou, H., “Versions of Schema for Object-Oriented Databases”, *Proceedings of the 14th International Conference on Very Large Databases (VLDB)*, Aug./Sept. 1988, pp. 148-159
- [6] Kim, W. *et al.*, “Architecture of the ORION Next-Generation Database System”, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, pp. 109-124
- [7] Khoshafian, Setrag, “Object Oriented Databases”, *John Wiley & Sons*, c1993
- [8] 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
- [9] Lautemann, S. E., “A Propagation Mechanism for Populated Schema Versions”, *Proceedings of the 13th International Conference on Data Engineering (ICDE)*, April 1997, pp. 67-78
- [10] Lautemann, S. E., Eigner, P., Wohrle, C., “The COAST Project: Design and Implementation”, *Proceedings of the 5th International Conference on Deductive and Object Oriented Databases (DOOD)*, Dec. 1997, *Lecture Notes in Computer Science 1341*, pp. 229-246
- [11] Loomis, M. E. S., “Object Versioning”, *Journal of Object Oriented Programming*, Jan. 1992, pp. 40-43
- [12] Monk, S. & Sommerville, I., “Schema Evolution in OODBs Using Class Versioning”, *SIGMOD Record*, Vol. 22, No. 3, Sept. 1993, pp. 16-22
- [13] Monk, S., “A Model for Schema Evolution in Object-Oriented Database Systems”, *PhD Thesis, Computing Department, Lancaster University*, 1993
- [14] “The O2 System - Release 5.0 Documentation”, *Ardent Software*, c1998
- [15] Odberg, E., “A Framework for Managing Schema Versioning in Object Oriented Databases”, *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, pp. 115-120
- [16] Odberg, E., “Category Classes: Flexible Classification and Evolution in Object-Oriented Databases”, *Proceedings of the Conference on Information Systems Engineering (CAiSE)*, *Lecture Notes in Computer Science 811*, pp. 406-420
- [17] Odberg, E., “A Global Perspective of Schema Modification Management for Object-Oriented Databases”, *Proceedings of the 6th International Workshop on Persistent Object Systems (POS)*, pp. 479-502
- [18] “Object Store C++ Release 4.02 Documentation”, *Object Design Inc.*, c1996

- [19] **“POET 5.0 Documentation Set”**, *POET Software*, c1997
- [20] 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
- [21] 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, *Lecture Notes in Computer Science 1460*, pp. 384-393
- [22] 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, Lecture Notes in Computer Science 1543*
- [23] 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
- [24] Sjoberg, D., **“Measuring Schema Evolution”**, *Technical Report No: FIDE/92/36*, Department of Computer Science, University of Glasgow, UK
- [25] Stroustrup, B., **“The C++ Programming Language”**, 3rd ed., Addison Wesley, c1997
- [26] **“Versant Manuals for Release 5.0”**, *Versant Object Technology*, c1997